The FAMOOS Object-Oriented Reengineering Handbook

http://www.iam.unibe.ch/∼famoos/handbook/

Holger Bär, Markus Bauer
Oliver Ciupke, Serge Demeyer
Stéphane Ducasse, Michele Lanza
Radu Marinescu, Robb Nebbe
Oscar Nierstrasz, Michael Przybilski
Tamar Richner, Matthias Rieger
Claudio Riva, Anne-Marie Sassen
Benedikt Schulz, Patrick Steyaert
Sander Tichelaar, Joachim Weisbrod

Version: October 15, 1999
(As Released to the general Public)

Editors of the Final Version: Stéphane Ducasse and Serge Demeyer
Previous Editors: Oliver Ciupke, Sander Tichelaar

This work has been funded by the European Union under the ESPRIT program Project no. 21975 (FAMOOS) as well as by the Swiss Government under Project no. NFS-2000-46947.96 and BBW-96.0015.

Contents

Preface

How to Read this Book

The book is organized into five parts:

I. Techniques. This first part describes various techniques that help during the reengineering life cycle. Source code metrics automatically measure properties of the software product and help in focusing the reengineering task by pin-pointing key classes of the system. Then the annotation of basic graphs with metrics reveals semantic structures that are hidden in plain source code and are generally helpful when presenting large amounts of data. Grouping is a way of building more abstract views that are based on the elementary, often overly detailed views from the source code in order to reveal higher-level problems. Finally, an approach to reorganisation is presented based on refactorings and design patterns.

II. Reverse Engineering

and III. Reengineering. The second and third parts form the core of the book. They consists of so-called *reengineering patterns*. Reengineering patterns capture tacit knowledge about when and how to apply reverse and reengineering tools and techniques as well as their implications. A reengineering pattern is somewhat like a design pattern. However, while a design pattern presents a solution to a design problem, a reengineering pattern relates two solutions (an existing solution and a target solution) via a process which transforms the one into the other. The reengineering patterns in the book tackle well known reverse and reengineering techniques often encountered in object-oriented programming.

IV. Tools. The third part contains a description of some of the tool prototypes that have been developed in the context of the Famoos project. Without tools, reengineering is an almost impossible task because of the huge amount of information comprising most legacy systems. To be able to exchange reengineering data between different tools, an information exchange model called FAMIX is proposed.

Background. The last part contains an introduction into software metrics and some discussions about the use of metrics when dealing with object oriented software.

Annotated Bibliography

Several good books exist today on how to improve the development of applications at several levels. We invite the reader to read these books in order to have a better overview of the field. This sections contains an annotated bibliography of material which is relevant to OO Reengineering. We did not aim for completeness, but rather selected information sources we have found interesting. Omission of any work does not imply that the work is less significant than those annotated here.

Software Engineering In General

- Both [SOMM 96] and [PRES 94] provide a broad overview of software engineering. Their books cover issues like reengineering and reverse engineering, CASE tools and metrics.
- [DAVI 95] provides lots of good practical advice on how to tackle software projects, some of which motivates work on reengineering.

Object-Oriented Engineering

- [GOLD 95] provides a decision framework for managing OO projects. Rather then imposing a particular software process or method, it tells you how you can built your own.

Conferences, Journals and Special Issues

- [CASA 97] is a special issue on object-oriented reengineering.
- [ARNO 92] is a book collecting various early papers on reengineering.
- [WATE 94], [WILL 96a] are more recent special issues on reverse and reengineering.
- Since 1994, there is a yearly conference on reverse engineering. It is called WCRE (*Working Conference on Reengineering*). The proceedings from 1995 onwards are published by IEEE Computer Society Press.
- ICSM (*International Conference on Software Maintenance*) and EuroMICRO (*Software Maintenance and Reengineering*) are other conferences focussing more on reengineering and maintenance. Their proceedings are also published by IEEE.
- The *Journal of Software Maintenance – Research and Practicve* is a journal dedicated to software maintenance and published bi-mothly by Wiley and Sons.

Metrics

- [FENT 97] is the seminal work on metrics but does cover very little on object-oriented metrics.
- [HEND 96] provides an overview of the state of the art in object- oriented metrics.
- [LORE 94] is a pragmatic handbook on how to use metrics to check object-oriented source code.

Object-Oriented Design

- [RIEL 96] presents a list of object-oriented design heuristics using C++.
- [MEYE 97] elaborates on Design by Contract.
- Design patterns are discussed in many books, most notably in [GAMM 95] and in proceedings of the different PLoP conferences.
- [LAKO 96] describes issues in building large scale (C++) systems including design considerations such as layering and more practical issues such as finding efficient include structures.

Information Exchange (Meta-Meta Models)

- CDIF (CASE data interchange format). See http://www.eigroup.org/
- MOF (Meta-Object Facility) and XMI. See http://www.omg.org/

Idioms

The books that follow contain practical information on exploiting programming language features to write good code.

- [BECK 97] contains a set of idioms related to the SMALLTALK language. The main focus of the book is to show how to write code that communicate its intent.
- [MEYE 98, MEYE 96] Meyer on the one hand focusses very much on the specific issues of C++ and explains complicated concepts such as the const mechanism in detail, or how to replace the default memory manager. On the other hand, Meyer also explains general OO concepts, like multiple inheritance, and its pitfalls in C++.
- [COPL 92] Coplien strives to teach C++ *fluency* by well known idioms like the orthodox canonical class form. He shows examples of how C++ can be used in a functional style. Some of the desing idioms presented in this book have been later rewritten into a pattern language.

UML, Object-Oriented Documentation

- [FOWL 97a] provides a fast introduction to UML including the notion of “perspectives” which is quite interesting from a reverse engineering point of view because it is a way to specify how a certain UML diagram should be interpreted (i.e., on a Conceptual, Specification or Implementation level).
- [BOOC 98], [RUMB 99] provide a good user reference and language reference for UML.
- [JOHN 92],[ODEN 97] present how patterns can support the documentation of a frameworks.
- [BROW 96], [WUYT 98], [PREC 98] present some possible approaches to support design patterns extraction.
- [FLOR 97] shows how design patterns can be supported at the development environment level.
- [STEY 96] presents *Reuse Contracts* as a way to document frameworks for evolution.
- [WINS 87] presents some discussion about variety of composition relationships.

Refactoring and Code Smells

- [FOWL 99] summarises practical experience with refactorings and code smells.
- The Ph.D. work of Opdyke [OPDY 92] on Refactoring resulted in a number of papers describing incremental redesign performed by humans supported by refactoring tools [OPDY 93], [JOHN 93].
- [ROBE 97a] describes the Refactoring Browser, a SMALLTALK tool that represents the state of the art in the field is described in and can be obtained from http://st-www.cs.uiuc.edu/.
- Both Casais [CASA 91], [CASA 92], [CASA 94], [CASA 95a] and Moore ([MOOR 96]) report on tools that optimise class hierarchies without human intervention. Schulz et al. illustrate the feasibility of refactorings on a subset of C++ [SCHU 98a].
- There exists a web-page discussing ”code smells”, i.e. suspicious symptoms in source code that might provide targets for refactoring. See http://c2.com/cgi/ wiki?CodeSmells

Reverse and Reengineering Taxonomy

- [CHIK 90] (reappeared in [CHIK 92]) provides a reverse and reengineering taxonomy. Unfortunately, it does not cover OO specific issues like refactoring. http://www.tcse.org/ revengr/ taxonomy.html

Organisations

- IEEE Computer Society’s Technical Committee on Reverse Engineering. See http://www.tcse.org/ revengr
- The Reengineering Forum (an industry association). See http://www.reengineer.org/

Chapter 1

The Need for Object-Oriented Reengineering

Reengineering legacy systems has become a vital matter in today's software industry. In the past few years, most of the reengineering efforts were focussed on systems written in traditional programming languages such as COBOL, Fortran and C. But recently an increasing demand for reengineering object-based systems has emerged. This recent evolution is not caused by failure of the object-oriented paradigm. Rather, it illustrates that the mere application of object-oriented techniques is not sufficient to deliver flexible and adaptable systems. This is due to a number of obvious problems:

- *Lack of experience*. It requires several years of experience to fully exploit the potential of the object-oriented paradigm. Such experience is often built up during the initial stages of a project, at the time when the most crucial parts of the system are implemented.
- *Hybrid programming languages.* The use of hybrid languages –like C++ and Ada–, combined with a "learn on the job" approach, prevents programmers from making the necessary paradigm shift.
- *Technology expansion.* Legacy systems could not benefit from emerging standards (e.g., UML, CORBA), technological advancements (e.g., design patterns, architectural styles) and extra language features (e.g., C++ templates, Ada inheritance).

These problems are accidental in nature: given proper training and sufficient tool support they will eventually be resolved. So why should one worry about object-oriented reengineering, since within a few years there won't be any more object-oriented legacy systems? In fact there a more fundamental problem.

The law of *software entropy* dictates that even when a system starts off in a well-designed state, requirements evolve and customers demand new functionality, frequently in ways the original design did not anticipate. A complete redesign may not be practical, and a system is bound to gradually lose its original clean structure and deform into a bowl of "object-oriented spaghetti" [WILD 92], [CASA 98], [BROW 98].

Many of the early adopters of the object-oriented paradigm have experienced such software entropy effects. Their systems are developed using object-oriented design methods and languages of the late 80s and exhibit a range of problems that prohibits them meeting the evolving requirements imposed by their customers. Their systems have become overly rigid thus compromising thier competitive advantage and as a consequence object-oriented reengineering technology has become vital to their business.

1.1 The FAMOOS Project

The need for object-oriented reengineering technology has been recognised by two of the leading European companies, namely Daimler-Benz and Nokia. Together with the University of Berne, Forshungszentrum

Informatik, SEMA Spain and Take5 they started a research project –named FAMOOS [1] – to investigate tools and techniques for dealing with object-oriented legacy systems.

The handbook you are reading right now is one of the main results of the FAMOOS project. It collects techniques and knowledge on the problem of software evolution with a special emphasis on object-oriented software. Most of the subject matter is not "new" in the sense that it represents new discoveries. Rather the handbook regroups much of the knowledge about redesign, metrics and heuristics into a single work that is focused on object-oriented reengineering.

1.1.1 Case Studies

All the techniques described in the handbook have been verified on six industrial case-studies, ranging from 50.000 lines of C++ up until 2,5 million lines of Ada. Figure 1.1 provides a quick overview of all of the FAMOOS case studies.

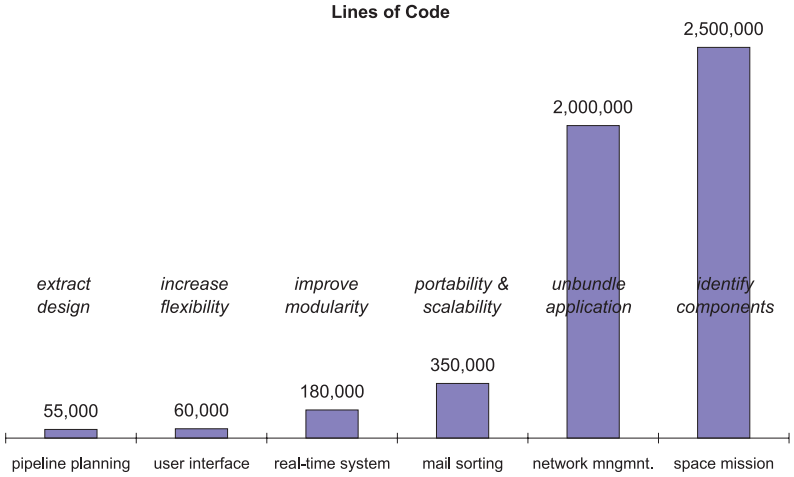


Figure 1.1: Overview of the FAMOOS Case Studies

- *Pipeline Planning.* The system supports the planning of liquid flow in a pipeline between multiple stations. The reengineering goal was to extract design from source-code, in order to reduce the cost of implementing similar systems, probably in other languages. The system is written in C++ and is a candidate for being rewritten in Java or Smalltalk.

- *User Interface.* This software provides graphical representations of telecommunication networks to telecom operators. The reengineering goal was to increase the flexibility of the software, i.e. improve its portability, facilitate addition of functionality and enhance tailorability towards customers. The system is written in C++.

- *Real-time System.* This software provide operating system features for embedded real-time controlling of hardware. The reengineering goal was to improve modularity for gaining shorter edit-compile-run cycles. The system is written in a mixture of C and C++.

- *Mail Sorting.* A control system for machines sorting mail envelopes. The software is highly configurable, to deal with the different ways countries over the world handle letters. The software itself is based on an internally developed distributed architecture which hindered the future evolution. The

[1] If you want to read more about the FAMOOS project and its results, we suggest to browse the web-sites offered by the respective project partners: http://dis.sema.es/ projects/ famoos/; http://www.iam.unibe.ch/ famoos/; http://www.fzi.de/ prost/

reengineering goal was to investigate how new technology could improve the portability and scalability (e.g. CORBA, Java, HTML). The system is written in a mixture of C and C++.

- *Cellular Network Management.* This case-study concerned a management system for digital networks. The main goal of the reengineering project was to unbundle the application, i.e. split the system into sub-products that can be developed and sold separately. The system is written in a mixture of C and C++.
- *Space Mission Management.* A set of applications that in different combinations form systems to support the planning and execution of space missions. The reengineering goal was identify components in order to improve reliability and facilitate system maintenance. The system is written in Ada.

1.1.2 Reengineering Goals

From this list of case studies some interesting information can be learned. First of all, the goals and motivations for reengineering the software systems are quite diverse, yet some common themes emerge.

- *Unbundling.* Unbundle the software system into subsystems that can be tested, delivered and marketed separately.
- *Performance.* Improving performance is sometimes a goal and sometimes considered as a potential problem once the system is reengineered.
- *Port to other Platform.* Porting to other (user-interface) platforms, sometimes requiring overall changes to the system.
- *Design Extraction.* Always a necessary step in understanding the system; sometimes even an explicit reengineering goal.
- *Exploitation of New Technology.* This may range from new features of the programming language up until upcoming standards (CORBA and UML).

1.1.3 Architectural Problems

Besides the motivations for reengineering problems, the case studies experience recurrent problems that are perceived as key obstacles for achieving the stated reengineering goals. Solving these problems requires significant human intervention since it involves an intimate understanding of and considerable changes to the architecture of the legacy system.

- *Insufficient Documentation.* All of the case studies face the problem of non-existent, unsatisfactory or inconsistent documentation. Tools to document module interfaces, maintain existing documentation and visualise the static structure and dynamic behaviour are required.
- *Lack of Modularity.* Most of the case studies suffer from a high degree of coupling between classes / modules / sub-systems that hampers further software development (compilation, maintenance, versioning, testing). A solution will involve metrics to help detect such dependencies and refactoring tools to help in resolving them.
- *Duplicated Functionality.* In many of the case studies several modules implement similar functionality in a slightly different way. This common functionality should be factored out in separate classes / components, but tools are missing which help in recognising similarities and in restructuring the source code.

- *Improper Layering.* In a few case studies the user-interface code is mixed in with the ”basic” functionality, creating problems in porting to other user-interface platforms. A general lack of separation, or layering, is observed with regard to other aspects (distribution, database, operating system) in other case studies. In contrast to a lack of layering, one case study suffers from unnecessary layers. Overly layered modules resulted from each successive developer encapsulating the module with a new concept instead of revising it. This problem needs tool support for defining layers and subsequent correction of broken layers.

1.1.4 Code Clean Up

There are quite a number of problems that have to do with ”code clean up”. Many of these problems arise from the lack of familiarity with the new object-oriented paradigm. But several years of development with sometimes geographically dispersed programming teams that change over time exacerbate these problems. Since they involve behaviour preserving restructuring of code only, those problems could be identified and repaired almost mechanically.

- *Misuse of Inheritance.* Inheritance is used as a way to add missing behaviour to one superclass. This is a often a result of having a method in a subclass being a modified clone of the method in the super-class.

- *Missing Inheritance.* In some cases, programmers have duplicated code instead of creating a subclass. In other parts, long case statements that discriminate on the value of a variable are used instead of method dispatching on a type.

- *Misplaced Operations.* Operations on objects were defined outside the corresponding class. Sometimes this was necessary in order to patch ”frozen” designs.

- *Violation of Encapsulation.* This was observed in extensive use of the C++ friend mechanism. Also, software engineers rely on the strong typing of the compiler to ensure certain constraints, and afterwards use typecasts to circumvent the safety-net. In some cases this leads to redundant type definitions which contaminate the name space.

- *Class Misuse.* This problem has been named ”C style C++”, although it is observed in Ada as well. It refers to the usage of the classes as a structuring mechanism for namespaces only. Sometimes this is necessary to interface with external non object-oriented systems.

1.1.5 Requirements

Last but not least, the case studies impose a number of constraints on the techniques and heuristics presented in this book.

- *Language Independent.* All material in this handbook is applicable on all major object-oriented languages, in particular C++ and Ada, Java and Smalltalk.

- *Scalable.* Some techniques and heuristics scale better than others. Rather than restricting ourselves to those techniques that can deal with small as well as large systems, we choose to specify for every technique the scale of systems it can be applied upon.

- *Tool Support.* There is a heavy emphasis on available tool support for all techniques covered in the book.

1.2 Basic Terminology

Before diving in the specific solutions for object-oriented reengineering, it is useful to agree on some terminology. We rely on the taxonomy of Chikofsky and Cross which is well-accepted within the reengineering community [CHIK 90]. For terminology specific to the object-oriented paradigm, we draw upon the design pattern book [GAMM 95].

- *Reverse engineering.* Originally used for the process of analysing hardware to discover its design, the term refers to the process of recovering information from an existing software system. In general reverse engineering seeks to recover information at a higher level of abstraction such as design information from code. Reverse engineering does not involve modifying the software system: it may be done as a stage in the reengineering process (model capture), as part of an effort to document the system, or as an attempt to extract reusable components from the software.
- *Forward engineering.* Refers to the usual process of software engineering: moving from requirements to high-level design, to progressively lower design levels and to implementation. While it may seem unnecessary to introduce a new term, the adjective ”forward” has come to be used where it is necessary to distinguish from reverse engineering and reengineering.
- *Reengineering.* Reengineering is the modification of a software system which in general requires some reverse engineering to be done. That is, reengineering requires that we first recover a view of the system at a higher level of abstraction than the code itself, then make changes to this view and implement these changes at the code level again. Simplistically, reengineering thus involves moving from code to model (reverse engineering), making modifications to the model, and then moving to ”better” code (forward engineering).

  There is some discussion as to whether or not reengineering involves a change in the functionality of the system – what it does for the user – since practically speaking reengineering almost always modifies the existing behaviour of the system, and indeed is usually motivated by a need to meet new requirements.
- *Restructuring.* Restructuring refers to transforming a system from one representation to another while remaining at the same abstraction level. At implementation level, this usually means changing the code structure without changing the semantics. However, even if the semantics are not changed at implementation level, restructuring might affect higher levels of abstraction (changing design vocabulary without affecting the implementation).
- *Refactoring.* Refactoring is restructuring within an object-oriented context. Refactoring involves tearing apart classes into special and general purpose components and rationalising class interfaces. The principle behind refactorings is that some relatively simple transformations (e.g., renaming a class, renaming a method, moving a method or attribute to another class) are combined into quite powerful semantic preserving transformations (i.e., componentise parts of a class, introduce a bridge design pattern).

1.3 The Reengineering Life-cycle

In this section we present the FAMOOS reengineering life-cycle. We regard reengineering as an evolutionary process consisting of the following six stages (see also figure 1.2):

1. Requirements analysis: identifying the concrete reengineering goals.
2. Model capture: documenting and understanding the design of a legacy system.
3. Problem detection: identifying violations of flexibility and quality criteria.

4. Problem analysis: selecting a software structure that solves a design defect.

5. Reorganisation: selecting the optimal transformation of the legacy system.

6. Change propagation: ensuring the transition between different software versions.

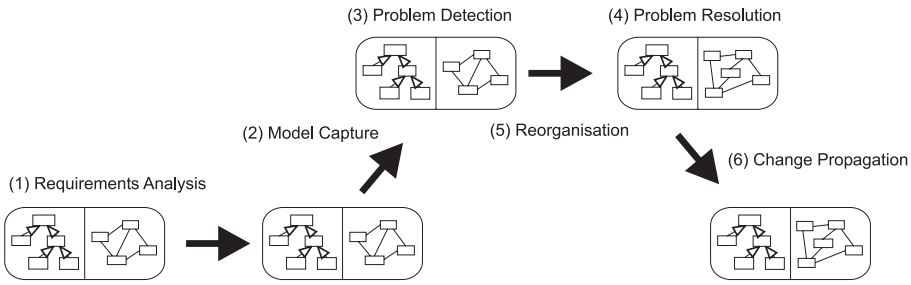


Figure 1.2: The Reengineering Life-cycle

Several iterations of these re-engineering stages might be needed before achieving a stable system with the desired degree of generality and adaptability.

Requirements Analysis. The specification of the criteria that the new, reengineered software must fulfill (for example, faster network performance).

Model Capture. In order to understand and to manipulate an object-oriented legacy system, it is necessary to capture its design, its architecture and the relationships between different elements of its implementation. A common problem in legacy systems is the lack of documentation. As a consequence, a preliminary model capture is often unavoidable, in order to document the system and the rationale behind its design. This requires reverse-engineering the legacy system to extract design information from the code.

Problem Detection. According to the reengineering requirements, problem areas within the legacy systems need to be detected. This requires methods and tools to inspect, measure, rank and visualise software structures. The problem areas have typically properties that deviate strongly form the properties as they are defined in the requirements. Detecting the problems with respect to flexibility requires a definition of these deviations (for example through thresholds on metrics). Problem detection can be based on a static analysis of the legacy system (i.e. analysing its source code or its design structure), but it can also rely on a dynamic usage analysis of the system (i.e. an investigation of how programs behave at run-time).

Problem Analysis. Upon detection of possible defects in the legacy system, software developers have to analyse them; that is, match detected problems against unmet requirements and understand how they concretely affect the software. Because applications are organised as intricate webs where classes, objects and methods may participate in various interactions, a detected problem may have to be decomposed into elementary sub-problems. A selection follows of appropriate target software structures - such as design patterns - that impart the software with the desired flexibility and functionality. A combination of such structures may be necessary to handle the particular design defect at hand. A prerequisite for problem analysis is an identification and specification of software structures to serve as the targets of reengineering, and a classification that allows to look for target structures corresponding to particular flexibility criteria or functional requirements.

Reorganisation. This phase of reengineering consists in physically transforming software structures according to the operations selected previously. This requires methods and tools to manipulate and edit software systems, to reorganise and recompile them automatically, to debug them and check their consistency, and to manage versions of software.

Change Propagation. The process of establishing a revised system throughout a corporate software environment. This might involve reengineering methodology that supports dissemination of improvements in more than one step.

Part I

Techniques

Chapter 2

Techniques

To reengineer and reverse engineer a software system one needs a range of techniques. This chapter provides a summary of the techniques investigated within the FAMOOS project.

Metrics are definitively appealing as support for understanding huge systems [DEME 99a], [DEME 99b], [BAUE 99]. Section 2.1 presents a list of the principal metrics and discussed their possible use and applicability.

Program Visualization is well suited to help to understand huge systems [DEME 99b], [LANZ 99]. Section 2.2 presents how the combination of simple graph layouts and metrics gives a quick means to understand and analyse an application.

Abstracting from the code contributes to the understanding of the system and can help to detect certain flaws [CIUP 99]. Section 2.3 presents how grouping entities at another level of abstraction supports a first analysis of the system.

Refactoring is now a well-known technique that helps behavior transforming code change [JOHN 93], [ROBE 97a], [FOWL 99]. Section 2.4 will focus more on advanced techniques based on design pattern based trasnformations.

2.1 Metrics

Author: Markus Bauer

2.1.1 Introduction

What are software metrics? – Formally, they measure certain properties of a software project by mapping them to numbers (or other symbols) according to well-defined, objective measurement rules. The measurement results are then used to describe, judge or predict characteristics of the software project with respect to the property that has been measured. Usually, measurements are made to provide a foundation of information upon which decisions about software engineering tasks can be both planned and performed better.

Although software metrics can be used to measure properties of the software development processes as well as organisations that produce software, we will only deal with software product metrics. These metrics measure properties of the source code of a software project and are the most interesting ones within the context of reengineering. [1]

To illustrate the concept of a software product metric, consider one of the most famous software metrics, *Lines of Code*. This metric measures the size of a piece of source code. We use this example to introduce the format which we will use to describe the metrics in this text: Each metric is presented with an acronym and its full name; a scope, explaining what entities of the software system are being measured (the system as a whole, a class,...); a category (the metrics in this text can be grouped into certain categories, see below); a (detailed) description, defining the metric and thus giving the measurement rule; related metrics that measure the same properties and references pointing to the original paper, where the metric has been defined first.

LOC – Lines Of Code.

Scope	System, Class, Method
Category	Complexity
Description	*Measures the size of a piece of source code by counting its lines.* Since the size of some source code can be seen as an indicator of its complexity, *LOC* is often used as a complexity metric or as an indicator on how much effort required to implement that piece of code. The line counting is usually done with respect to a certain coding standard which defines precisely what constitutes a line of code in a particular programming language. This is necessary for obtaining comparable, well-defined measurement results.
See also	–
References	[HUMP 97] provides a good discussion of all aspects related to *LOC*.

Why are software metrics important when reengineering (object oriented) legacy systems? – Software metrics support numerous reengineering tasks, because they help to focus reengineering efforts. They aid in forming an initial understanding of the legacy system and can often uncover hints about design flaws that that are obstructing the modification and extension of the system. Metrics lend themselves we to automatization and with appropriate tools they can provide easy access to meaningful information about the source code without requiring you to read through all the source code by hand. Instead you can use the information to make a more efficient study of the source code based on the points of interest indicated by the metrics results.

[1] Note, however, that process metrics or metrics that measure resources of an organisation should still be applied in reengineering projects to support the project management, but this is beyond the scope of this text.

The next section of this text gives an overview over some important object-oriented software metrics and explains some basic properties that can be measured by them. This provides the background needed to present how metrics can be used during reengineering tasks through some some typical usage scenarios based on some of these metrics. We believe that learning how metrics are applied in these usage scenarios will illustrate ideas on how to use metrics in your own reengineering projects.

2.1.2 Some Important Metrics

In this section, we present some object oriented software metrics, that have proven to be particularly useful[2]. These metrics fall into several categories depending on thes aspects of a system they measure. We have identified the following categories: *complexity* metrics, *coupling* metrics, *cohesion* metrics and *inheritance tree* metrics.

2.1.2.1 Complexity Metrics

Complexity metrics measure the *complexity* of an entity of a system. The metrics presented here measure the complexity of a class. By the term complexity of a piece of software or source code, we usually try to describe how much effort has to be spent by a software engineer to understand, write or modify that piece of software – code that is difficult to read and understand is considered as complex. Since measuring the complexity directly is not possible (since we would have engineers read the code and check how much time they needed to understand it), we use some metrics to estimate that complexity. The metric *LOC* (p. 22), mentioned above, is an example of such a complexity metric.

One of the well established metrics to measure the complexity of a class is:

WMC – Weighted Method Count.

Scope Class

Category Complexity

Description *Measures the complexity of a class by adding up the complexities of the methods defined in the class.* Thus,

$$WMC = \sum_{i=1}^{n} c_i$$

where c_i denotes a complexity measurement of method i.

Complexity measurements for methods are usually given by code complexity metrics like *LOC* (p. 22) or the *McCabe cyclomatic complexity*. The McCabe cyclomatic complexity measures the complexity of some code by taking into account the decision structure of the code, i.e. code that contains a lot of loops or *if-then-else*-constructs is considered more complex.

See also *NOM* (p. 24) is a special case of this metric – all method complexities are assumed to be 1.

References [CHID 94], [CHUR 95], [ETZK 99]

[2] Chapter 20 contains a survey and critiques on a large amount of metrics, we used that survey to select the metrics presented here. Additionally, we recommend [LORE 94] as a good textbook on metrics.

A special case of *WMC* (which is very simple to compute) is:

NOM – Number Of Methods.

Scope	Class
Category	Complexity
Description	*Measures the complexity of a class by counting the number of methods defined in that class.*
See also	*WMC* (p. 23)
References	[HEND 96]

Obviously, complexity metrics play an important role when reengineering software systems: classes with high complexity measurements are difficult to understand and consequently difficult to change. For details, check the scenarios in sections 2.1.3.1-2.1.3.4.

2.1.2.2 Coupling Metrics

Another important aspect when dealing with a legacy system is the coupling between classes. A class is *coupled* to another class, if it depends on (or knows) that class, for example by accessing variables of that class, or by invoking methods from that class.

DAC – Data Abstract Coupling.

Scope	Class
Category	Coupling
Description	*Measures coupling between classes that results from attribute declarations.* *DAC* counts the number of abstract data types defined in a class. Essentially, a class is an abstract data type, therefore *DAC* reflects the number of declarations of complex attributes, i.e. attributes that have another class of the system as a type.
See also	*RFC* (p. 24), *CBO – Coupling Between Objects* [CHID 94], *NIV – Number of Instance Variables* [LORE 94]
References	[LI 93], [HITZ 95], [HITZ 96a]

The following metric is a coupling metric as well; however, the complexity of the class affects the measurement, thus it cannot be considered as a pure coupling metric.

RFC – Response Set For A Class.

Scope	Class
Category	Complexity, Coupling
Description	*Measures complexity and coupling properties of a class by evaluating the size of the response set of the class*, i.e. how many methods (local to the class and methods from other classes) can be potentially invoked by invoking methods from the class. More formally, RFC for a class C is defined as $RFC = \|RS\|$, where the response set RS is given by

$$RS = M \cup \bigcup_{m \in M} R_m$$

M is the set of methods defined in C and R_m is the set of methods called by method $m \in M$.

See also *DAC* (p. 24), *MPC – Message Passing Complexity* [LI 93]

References [CHID 94]

Why are we interested in the coupling between classes? – Classes, that are tightly coupled cannot be seen as isolated parts of the system. Understanding or modifying them requires that other parts of the system must be inspected as well. Conversely, if other parts of a system get changed, classes with high coupling measurements are more likely to be affected by these changes (see scenario in section 2.1.3.3). Additionally, classes with high coupling tend to play key roles in the system, making them a good starting point when trying to understand an unfamiliar legacy system (see scenario in section 2.1.3.1).

2.1.2.3 Cohesion Metrics

The *cohesion* of a class describes how closely the entities of a class (such as attributes and methods) are related. Often, cohesion is measured by establishing relationships between methods of the class in the case where the same instance variables are accessed. A useful metric measuring this property is:

TCC – Tight Class Cohesion.

Scope Class

Category Cohesion

Description *Measures the cohesion of a class as the relative number of directly connected methods*, where methods are considered to be connected when they use at least one common instance variable.

More formally *TCC* for a class C is defined as follows: Let

$$NDC = |\{(m,n) \mid \text{ methods } m,n \text{ access a common instance variable}\}|$$

be the number of connected methods and $NPC = \frac{n(n-1)}{2}$ the possible number of connected methods, then

$$TCC = \frac{NDC}{NPC}$$

See also *LCOM – Lack of Cohesion On Methods* [CHID 94]

References [BIEM 95], [HITZ 95], [HITZ 96a], [ETZK 98]

Cohesion is an important concept: good object oriented design styles usually require that classes have high cohesion, since they should encapsulate concepts that belong together. Classes with low cohesion often represent violations to a flexible, extensible or reusable design. All of these are issues that must be dealt with during reengineering projects. The scenario in section 2.1.3.2 further deals with this issues.

2.1.2.4 Inheritance Tree Metrics

A basic concept that is central to object-oriented systems is *inheritance*. Through inheritance, relationships between objects can be modelled (for example the *is-a* relationship). Moreover, inheritance is often used to allow for some reuse of existing classes. Accordingly, measuring properties of the inheritance tree of a system often gives interesting results. Some simple metrics relating to the inheritance tree and its layout are:

DIT – Depth in Inheritance Tree.

Scope	Class
Category	Class Hierarchy
Description	*Measures the depth of a class in the system's inheritance tree*. The *DIT*-value for a class C is defined as length of the longest path in the inheritance tree from the root class of the system to C.
See also	—
References	[CHID 94]

NOC – Number Of Children.

Scope	Class
Category	Class Hierarchy
Description	*Counts the number of children (direct subclasses) of a class.*
See also	*NOD* (p. 26)
References	[CHID 94]

NOD – Number Of Descendants.

Scope	Class
Category	Class Hierarchy
Description	*Counts the number of descendants (direct and indirect subclasses) of a class.*
See also	*NOC* (p. 26)
References	[TEGA 95]

Obviously, the inheritance metrics presented above may be used to measure a special case of coupling – the usage of classes through inheritance relations. For example, classes with low *DIT* values and high *NOC*- or *NOD*-values are classes that affect a lot of other classes, because they are (direct or indirect) super classes to them. Changes to such classes are likely to require changes in the subclasses. We believe, however, that inheritance based properties of a system are more easily understood through visualising the inheritance tree.

Recently, more sophisticated metrics have been defined that measure the amount of reuse in an inheritance tree. Because of their complexity, these metrics are outside the scope of this text, we refer you to chapter 20 for a description of these metrics.

2.1.3 Usage Scenarios

In this section, we present some scenarios illustrating how metrics can be applied successfully in software reengineering projects.

2.1.3.1 Get a Basic Understanding of the System

Usually, one of the initial steps when reengineering a legacy system is to aquire a basic understanding on how the system works and how it is structured. However, the documentation alone is typically insufficient. Therefore some analysis of the system's source code is required. This is *(model capture)* and metrics can provide valuable help during this task.

A good way to start model capture is to find out which parts (i.e. classes) implement the key concepts of the system. A technique to do this is described in [BAUE 99]: Usually, the most important concepts of a system are implemented by very few *key classes*[3], which can be characterised by the following properties: Key classes *manage* a large amount of other classes or *use* them in order to implement their functionality, thus they are tightly *coupled* with other parts of the system. Additionally, they tend to be rather *complex*, since they implement much of the system's functionality.

Based on this observation, it is straightforward to identify these key classes by using both a complexity metric like *WMC* (p. 23) and a coupling metric like *DAC* (p. 24). Figure 2.1 shows a diagram that can be used for such an analysis – the classes of the legacy system are placed into a coordinate system according to their complexity and coupling measurements. Classes that are complex and tightly coupled with the rest of the system fall into the upper right corner and are good candidates for these key classes. To understand how the legacy system works we should thus concentrate on understanding these classes and their interactions by studying their source code.

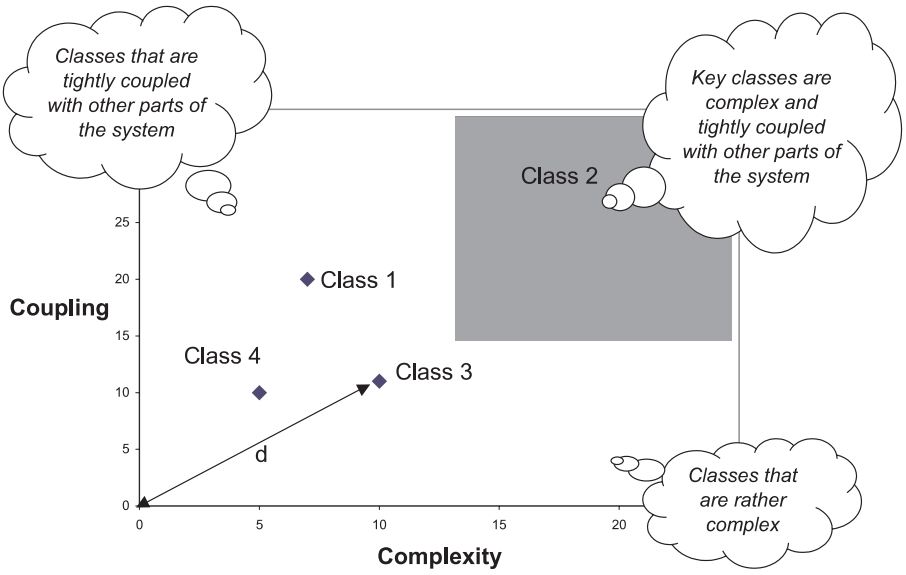


Figure 2.1: Finding the key classes in a legacy system

2.1.3.2 Find Violations of "Good" OO Design

Though reengineering projects are usually started in order to make the legacy system satisfy additional functional or non-functional requirements, general improvements to the software are often desired as well.

[3] Case studies in [BAUE 99] have shown that about 10 % of the classes of a system can be considered as key classes

One way to achieve such general improvements consists in finding violations of a ”good” object oriented software design *(problem detection)*. Unfortunately, there is no consensus on what ”good” design really is, however, some guidelines and principles exist that are considered helpful to achieve an understandable, flexible and extensible software design. Metrics are particularly suitable to check, whether the legacy system adheres to such design principles or to find violations of them.

A (very basic) principle in object oriented software engineering states that a class should implement *one single concept* (of the application domain). Some violations of that principle can be detected by using software metrics if we make the following assumptions:

- A class that implements more than one concept, has probably *low cohesion* measurements, since these concepts can be implemented separately.
- A class that by itself does not implement one concept, i.e. the implementation of the concept is distributed among many classes, is probably *tightly coupled* to other classes.

Therefore, by applying cohesion metrics like *TCC* (p. 25) and coupling metrics like *DAC* (p. 24) or *RFC* (p. 24) to the legacy system, possible violations of the principle ”one class – one concept” can be found. These classes tend to have either low *TCC*-values or high *DAC*-, *RFC*-values.

For example, classes that have very low *TCC*-values, can often be split. Sometimes this leads to a more flexible design, since the two separate classes are easier to understand and are more reusable. Low *TCC*-measurements can as well point to classes that have not been designed in an object-oriented flavour at all – these classes are not implementing a self contained object from the application domain, they just group methods together, acting as a module. In a similar manner other design principles can be checked and violations can be detected by using metrics [RIEL 96].

However, we should be aware of some difficult issues, when applying metrics for such problem detections: It is difficult to specify thresholds for the measurements, i.e. values, which classes adhering to a ”good” design should fulfill. Additionally metrics can produce ”false alarms”. They can label classes as being problematic, but there may well be a reason that these classes present untypical measurement values. Measurement results must always be taken with a grain of salt and problematic results should always be checked against the source code.

2.1.3.3 Identify Change Sensitive Parts

Whenever we make changes to an existing software system, it is likely that these changes will require further changes throughout the system since the entities of the system are interdependent. Changes in some parts of the system can produce a lot of work, if a lot of other parts depend on them, and, inversely, some parts probably change often during the evolution of a system, because they depend on lots of other parts and changes to the system are likely to affect them.

To make sure that the system does not misbehave after making some changes, we would be interested in localizing these *change sensitive* classes, i.e. classes that are most likely to be affected by changes to a system because the depend on lots of other parts. To do this, we can use coupling metrics like the *DAC* (p. 24). Classes with high *DAC*-values ”know” a lot of other classes and are therefore change sensitive. These classes should then be carefully examined and tested after modifications of the system.

2.1.3.4 Track the Evolution of the System

Most software systems *evolve* over the time, i.e. new functionality is added, extensions are made,.... This poses an important question: Does the quality of our system decrease during the evolution of the system? Do some reorganisation raise the quality of the system?

Metrics can be used to answer these question and to control the quality of the software. A lot of research work has been done in this area, see for example [LORE 94], [ERNI 96] or [DEME 99a].

The basic steps of using metrics for quality control are:

1. Establish quality goals for your software.
2. Decide on a set of metrics to check your software with respect to the quality goals.
3. Use the metrics to constantly monitor the quality while the system evolves.

A simple example: a very high level quality goal for a software system could be maintainability, thus, coupling measurements should not be high in order to ensure that changes to the system do not trigger changes throughout the system (see 2.1.3.3). Therefore, monitoring *DAC*-values can be promising. When a significant number of classes evolves to higher *DAC*-measurements, some refactorings of the system could be appropriate, to reduce coupling.

Another application of metrics when tracking the evolution of a system is to identify stable and unstable parts of your system (for details, see [DEME 99c]). Often, this can be interesting information: Stable parts can be declared as ”frozen” and can often be reused in other projects (i.e. factored out into frameworks), whereas unstable parts should be tested thoroughly.

2.1.4 Summary

In the previous sections we have seen what object oriented software metrics are, and how they can be applied in reengineering projects. We have illustrated that metrics are able to support model capture and problem detection phases. Figure 2.2 sums up our experiences with the metrics mentioned in this text and gives hints on how well these metrics are suited for model capture or problem detection.

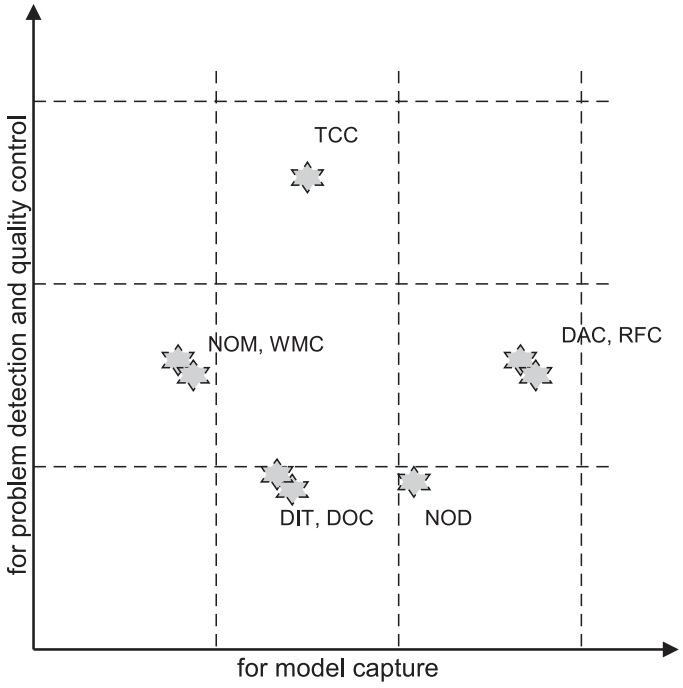


Figure 2.2: Applicability of the metrics

Reengineering projects can benefit from metrics in the following ways: Applied with well documented scenarios (as given in section 2.1.3), they make reengineering tasks more organised and focused. They provide an abstraction mechanism from the huge amount of source code of the legacy system, thus allowing you to concentrate your work on the important or critical parts of the system that have been identified by the measurements.

However, metrics can fail. They can point to wrong places in your source code, or you can even miss important classes of your system, because the measurements just do not ”catch” them. Therefore, metrics should be used with care, and you should always check with the system’s source code or make additional analyses to avoid drawing wrong conclusions.

2.2 Program Visualisation and Metrics

Author: Michele Lanza

> *"Continuous visual displays allow users to assimilate information rapidly and to readily identify trends and anomalies. The essential idea is that visual representations can help make understanding software easier."*
> [BALL 96]

Although the object-oriented paradigm lets programmers work at higher levels of abstraction than procedural models, the tasks of understanding, debugging, and tuning large systems remain difficult. This has numerous causes: the dichotomy between the code structure as hierarchies of classes and the execution structure as networks of objects; the atomisation of functionality - small chunks of functionality dispersed across multiple classes; and the sheer numbers of classes and complexity of relationships in applications and frameworks. The fields of scientific visualisation and program visualisation have demonstrated repeatedly that the most effective way to present large volumes of data to users is with a continuous visual fashion [PAUW 93].

In this chapter we list some properties that a graphical representation of source code should possess to be useful for reverse engineering. We then see in what respect our approach fulfils those requirements and include a short scenario to explain our approach. We also list some problems concerning the visualisation of metrics, colors and issues concerning interactivity.

The central point of this chapter is to show how we merge the concepts of program visualisation, metrics and interactivity. These three aspects are the cornerstones of this work. The concepts that are explained here have been implemented in a single tool called CodeCrawler, which we present in the next chapter.

2.2.1 Graphs for Reverse Engineering

In this section we list some features that in our eyes graphs for reverse engineering should have. We emphasise that we use the term graph in a very broad sense: often we mean its picture or graphical representation on screen or on paper and not necessarily its scientific definition.

- *Simplicity and Quality*. The first important prerequisite is that the generated pictures of a graph have to be relatively simple and easy to grasp. The main reason for that is that too much displayed information overloads the viewer's perception. This tends to backfire and causes an unwanted information loss. A secondary aspect is that simple graphs are also easily reproducible, while complex techniques like hyperbolic trees [LAMP 95] are affected by a considerable complexity which is hard to grasp and reproduce. Many approaches have been discussed as to how a software entity could be represented for program visualisation ([BALL 96, PAUW 93, KLEY 88] to name but a few). We think that a graphical representation of an object oriented entity should be easy to grasp and not make use of a specific dictionary of shapes which has first to be learned. A graph should be able to transmit useful information to the viewer at first sight.
- *Quantity*. We have to be able to select how much of the subject system we want to display and at what level of granularity. Thus, we should be able to zoom in and out of such a graph and reduce the amount of displayed information at will.
- *Colors*. Program visualisation can be supported by colors, because they can attract the eye to interesting hot spots, while other parts of the graph which look less colorful can be ignored by the viewer. Colors have often been used in program visualisation [RIVA 98]. While colors are a good way to attract the attention of the eye, the usage of too many colors in a graph is not advised, since this results in an optical overload for the viewer of the graph. We also advise against the use of color conventions which have first to be learned by the viewer, as this lessens the impact of the colors.

- *Scalability*. As reverse engineering is especially crucial in very large systems, a visualisation should be scalable and work if possible at any level of granularity. The number of displayed entities should not affect the quality of the graph.
- *Interactivity*. A very important aspect of graphs is not only their layout algorithm but also that they can provide interactivity to the user through direct-manipulation interfaces. Making a static display of nodes and trying to extract information from the graph has clearly defined limits, which we discuss below in Section 2.2.2.
- *Metrics*. Although intangible in the physical sense, software *has* size. It can be measured, especially in object-oriented code we can assign numerical values (metric measurements) to its entities. Although the concept of software is abstract and often exists only in the head of the programmer, we can measure it. Once we can measure it, we can assign a size to it and represent this size graphically. We think that metrics enrich the semantic value of a graphical representation of a software entity, and discuss this below in Section 2.2.4.

2.2.2 Interactivity

A graph which lacks interactivity has certain drawbacks:

1. The user can't produce new views starting from a part of the graph.
2. The user can't find out secondary information (e.g. he can't inspect the nodes or browse through their source code).
3. The user can't reduce the amount of displayed data by either removing nodes by hand or by filtering out nodes through algorithms.

Those limits can be overridden if the graph is interactive:

- If we produce a view on a system and one particular node is drawing our attention, we'd like to know more about this node and the entity that it is representing. So we should be able to know its name, to have a look at its properties, to zoom in into the node, to have a list of all nodes that have a relationship with this node, or even to have a look at the source code behind the node (suppose the node is a method).
- Starting from a part of the graph or from one single node we'd like to be able to generate new views without having to go through the whole graph generation procedure again. The viewer should be able to 'navigate' around the code travelling from one point of interest to the next.
- Sometimes the relationship edges in a graph make the whole graph look like a cobweb. We should be able to switch off edges and switch them on again on demand depending on nodes we selected, etc.
- Suppose we have displayed a graph with a lot of nodes and edges. One particular node is of interest to us. But since there are too many edges in the graph it's hard to see how many times and to which other nodes the node in question is connected. So the graph should also be able to provide a 'highlighting feature' where we can display on top of all edges and nodes the connections of the node in question. It is important to note here that compared to the previous point we don't want to reduce the complexity of the displayed graph. We just want to have a better view on it.

It is an important point we are stating: The interactivity of a graph is *not just a nice feature* but one of its *most important aspects*.

2.2.3 The Use of Layout Algorithms

Perhaps the most difficult aspect of showing software through graphs involves the graph layout problem. The nodes and edges of the graph must be positioned in a pleasing and informative layout that clearly shows the underlying graph's structure. Many techniques have been proposed for laying out arbitrary graphs. Unfortunately, in practice, drawing informative graphs is exceedingly difficult, particularly for large systems. The resulting graphs, even when drawn carefully, are often too busy and cluttered to interpret [BALL 96].

The opposite case can also be true: sometimes elaborate layout algorithms can't ameliorate the user's perception or can do that only at the cost of algorithm complexity: There are various (and sometimes very complex) techniques to display a tree graph, but in the end it's still just a tree.

However, we don't want to minimise the importance of complex layout algorithms, on the contrary: we believe they could bring many more benefits than drawbacks. Good layout algorithms just were not part of the constraints of this work. But it is certainly a very promising field of research in this context. The layout algorithms used in this work are discussed in detail in Section 2.2.6.

2.2.4 The Use of Metrics in Graphs

In [BALL 96] the following statement is made: "Software is intangible, having no physical shape or size. Software visualisation tools use graphical techniques to make software visible by displaying programs, program artifacts, and program behaviour."

It is obvious that everything regarding metrics possible through their graphical display is also possible by just calculating and analysing the metric measurements. So the question arises why we should have a graphical display of them, since the information sought is in the metrics themselves. But in the same way one could think to listen to music by just reading the partiture of a song instead of using the sense normally designed for that (the hearing)[4]. *What changes is the perception and the impact of what is perceived.*

Our Idea. The whole concept is fairly easy: we map metric measurements of software entities on their graphical representation on the screen. As we said before we chose the entities to be represented by rectangles. Rectangles have a certain width and a certain height. They can be filled with a color. Their position can also bear a certain amount of information.

With this approach, in a two-dimensional graph consisting of nodes and eventually edges between the nodes, up to five metrics can be assigned to a node and rendered visually at the same time. These are:

1. The X coordinate of the position
2. The Y coordinate of the position
3. The width
4. The height
5. The color shade

This concept is rendered clearly in Figure 2.3, where we see where the metrics can be applied on a node.

Not every graph can make use of five the metrics at the same time. In a graph that does not have an origin (which defines an absolute coordinate system) it makes no sense using two metrics for the position of the

[4] A short comment on perception: the size of software can be seen through other means: if we scroll through the source code of a very large class, we probably have to either move the mouse or press some keys on the keyboard to scroll on. This physical act of scrolling can also transmit size and complexity to the viewer.

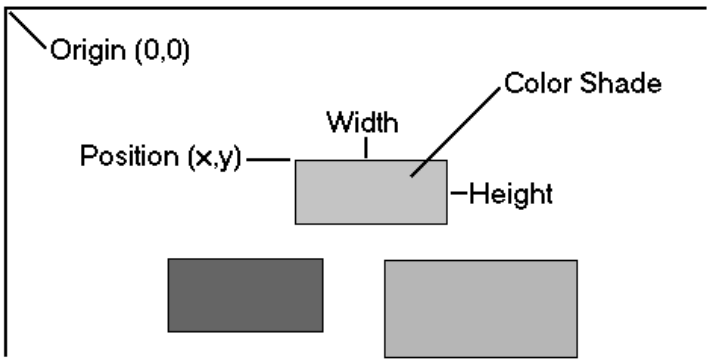


Figure 2.3: An example of nodes and their possible metrics.

nodes. A good example for such a graph is a tree graph, where the position on the nodes is implicitly defined by the logical position of the nodes in the tree. Another property which came up during our experiments was that sometimes the multiple use of the same metric (for example if we choose the same metric to reflect width and height) can emphasise certain parts of the graph and render them more clearly for the viewer.

2.2.5 A Concrete Graph Specification.

In our approach a *concrete graph*, this means the resulting displayed graph, is the combination of four factors :

1. **The Graph Type**. Its purpose is to render a certain aspect of a system: a tree graph is good for displaying hierarchical information, a circle for communication, a confrontation graph for dependencies, etc.

2. **The Layout Algorithm**. Starting from the original idea of the graph, variations refine the concrete display. The layout takes into account the following issues:

   - Display concerns (i.e. the fact that the complete graph should or not: fit into the screen, minimise the space used, sort the nodes according to certain criteria, etc.).
   - The entities and their relationships. This implies the choice of the represented entities (class, attribute and/or method) to be rendered as graphic elements and the logical link between the graphical elements and the metrics. For example in some graphs the position of the nodes reflects the size of the entities whereas in others this is the size of the node.

3. **The Metric Selection**. Once the layout algorithm stands, metrics are associated to the graph. This application depends on the specification of the previous step.

4. **The Interaction**. Since the goal of a graph is to support the reverse engineering of the application, the interaction that a user can perform should be specified. All the graphs support basic navigation functionality which allows one to access code elements. However, the interaction is refined for specific graphs, for example to walk through it, to highlight the edges, to zoom in/out, etc.

2.2.6 A Short Example

To make the whole idea of visualising software structures with the help of metrics a bit more understandable we included here a short example of our approach.

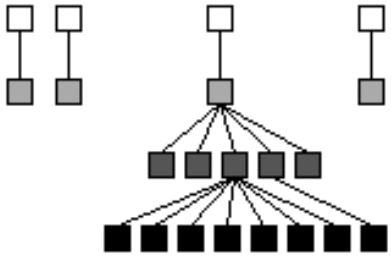


Figure 2.4: A simple inheritance tree.

Suppose we want to understand the inheritance hierarchy of a small system. The idea that comes up is to display the graph as a tree. The nodes in tree represent classes, the edges represent inheritance relationships.

The layout algorithm for displaying a tree is arbitrary, for reason of simplicity we chose a very simple one, which sometimes can make edges cross nodes, but it renders the whole concept nonetheless. Keep in mind that this layout part can also make use of very complicated algorithms for space optimisations, edge crossing reduction, etc.

Once we have displayed the tree as we see in Figure 2.4 we apply size and color metrics to the nodes. The use of position metrics is not possible here, as the position of the nodes is intrinsically defined by their logical position in the tree. As the nodes represent classes, we use class size metrics. The width and height of the nodes render the number of methods (NOM) while the color renders the number of attributes (instance variables).

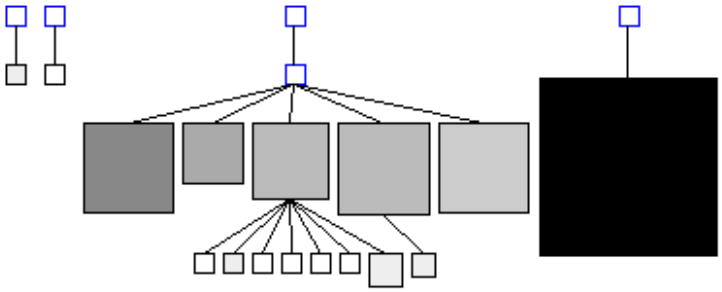


Figure 2.5: An inheritance tree that makes use of size and color metrics.

Once the tree is rendered as in Figure 2.5 we can start interacting with the graph. We can freely move nodes around, delete them, inspect them (i.e. browse the underlying classes), filter out parts, etc.

In fact, if we left out the interactive part, the amount of useful information that we could extract would be limited to the display in Figure 2.5.

Useful Graphs

This section is dedicated to the graphs which prove to be useful when it comes to the understanding of software systems and the detection of design problems using the approach discussed in this work. Although this may seem a little confusing, what in this chapter is called a 'useful graph' is not only its layout, but primarily the *combination of a layout with object-oriented metrics and the consequent extraction of information made by the viewer through interaction with the graph*.

The following is structured as follow:

- **Graph Structure.** Section 2.2.7 presents the structure we adopted to describe a useful graph. Every useful graph is presented using this format.
- **Case Studies.** Section 2.2.8 is a short presentation of the two case studies we chose to apply each useful graph on.
- **Layout Algorithms.** Section 2.2.9 presents the layout algorithms we selected.
- **Useful Graphs.** In section 2.2.10 we present the useful graphs divided into the 4 distinct groups: *class*, *method*, *attribute* and *class internal*. The names indicate which kind of entities are displayed in the graphs. Class internal treats the special case where methods and attributes are displayed at the same time.

2.2.7 Graph Structure

For each graph which we treat in this chapter, we discuss the following properties:

Graph: Indicates what type of graph and layout has to be chosen, and whether a sorting of the nodes has to precede the display.

Scope: At what granularity level the graph can be applied. We differentiate between *full system*, *subsystem* and *single class*. Sometimes the subsystems are indicated as a single inheritance hierarchy. We also indicate if the graph is language specific.

Metrics: We list five metrics in the following order: width metric, height metric, color metric, horizontal position metric, vertical position metrics. When we write a dash (-), this means that the metric should not be set. In case we write an asterisk (*) this means that the metric can be set freely. In the case of class internal graphs we repeat the five metrics twice, once for the method nodes and once for the attribute nodes.

General idea: We write what the graph is all about and what ideas lie underneath it. We also indicate what the user should be searching for in the graph.

Results: Here we present the results obtained after applying the graph on our case studies.

Possible Alternatives: We list a few alterations that could be made regarding the metrics, so as to obtain slightly different graphs, and list also some eventual interactions that could be applied on the graph to increase its usefulness.

Evaluation: Some statements about the advantages and drawbacks of the graph.

Application	Refactoring Browser	Duploc
Classes	166	123
Methods	2365	2382
Attributes	365	386

Table 2.1: An overview of the size of our case studies.

2.2.8 Case Studies

This section contains a short overview of the systems we used as case studies for this work. Basically we use them to test the graphs listed in the remainder of this chapter. We chose these two case studies for the following reasons:

- Availability. Both case studies are public domain and can be downloaded freely. With this point we can ensure that the results are reproducible.
- Size. We chose two case studies which can be termed as being of an *average size* and are representative of medium-sized standalone applications. We think that very small applications can't reflect results properly because the purpose of most graphs is coping with complexity, which in such cases is not necessary. On the other hand, if we had chosen very big applications, it would have been hard to present results in a concise manner, because many graphs can be applied in various areas and at various levels of granularity.
- Level of maturity. We chose one very mature application which has gone through some refactorings and redesigns, and another one which has been developed in a rush and which has yet to undergo its first redesign. We did this to see if the results of our experiments would differ and in what way they would do that.

Refactoring Browser. The Refactoring Browser is a widely used tool for the implementation of Smalltalk programs [ROBE 97a]. We took it as a case study because it is an application which has gone through several refactorings and redesigns and has been written by some very experienced programmers. This quality of implementation should thus be visible in such a system. It is a medium sized application as we can see in table 2.1.

Duploc. Duploc is a tool for the detection of duplicated code [RIEG 98]. Duploc was the first application written in Smalltalk by its developer, Matthias Rieger and has yet to undergo its first major redesign. Thus we expect it to have some of the flaws which new systems tend have, like oversized classes and methods, obsolete attributes, etc.

2.2.9 Graphs

This section is dedicated to the graphs and layouts we have selected to implement in CodeCrawler. We discuss the properties, advantages and drawbacks of each one of them. We include this here because they are mentioned throughout the remainder of this chapter.

We discuss the original idea of a graph and the scope of its applicability. Each graph has at least one possible kind of layout and we discuss it with a regard for the metrics that can be applied for that special layout. Sometimes a sorting of the nodes has an influence on the usefulness of a graph and we discuss that as well as the general pros and contras for each graph.

In Table 2.2 we have an overview of all graphs and their properties supported by CodeCrawler.

Graph Type	Metrics	Entities	Sort	Scope
Tree	3	C		Global
Correlation	5	CMA		Global- Local
Histogram	3	CMA	X	Global- Local
Checker	3	CMA	X	Global- Local
Stapled	3	CMA	XX	Global- Local
Confrontation	3 + 3	MA	X	Local

Table 2.2: CodeCrawler's graph layouts.

The 'Metrics' column specifies how many metrics can be rendered by the graph. 5 means that the a single node can render 5 metrics at the same time. $3 + 3$ means that two separate groups of entities and metrics can be defined. The 'Entities' column refers to the kind of entities the graph can be applied upon: C for class, M for method and A for attribute[5]. The 'Scope' column specifies if the graph can be applied to the complete (sub)system or only to some entities like a class or a method. The 'Sort' column indicates if a sorting of the nodes according to a certain metric measurement can enhance the usefulness of the graph in question.

[5]The limitation to these three types of entity is due to the current implementation of the Moose model. Future implementations of it may include supplemental entities.

2.2.9.1 The Tree Graph

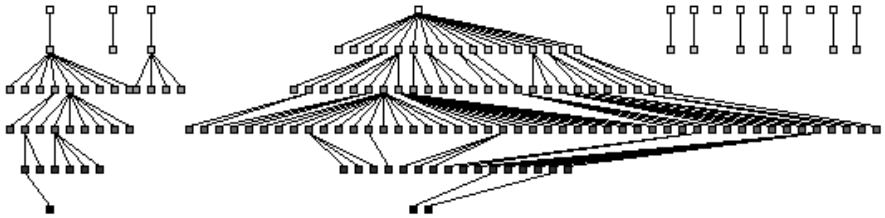


Figure 2.6: A tree graph of a system.

Overall Idea. A tree graph is useful for the display of hierarchical structures like inheritance hierarchies containing classes. The nodes represent classes, while the edges between the nodes represent inheritance relationships.

Scope. The scope of this graph ranges from very large systems to subsystems consisting of few classes. A requirement is that there is some usage of inheritance in the system. Otherwise the graph gets very flat and wide.

Layouts. We implemented three slightly different layout algorithms, which we simply called left, centered and right. Each one of them is based on recursion.

Metrics. The number of possible metrics that can be applied is 3. The two position metrics cannot be used, as the position of the nodes is defined by the layout algorithm. However, a virtual fourth metric is present, HNL. It is rendered by the layout algorithm through the vertical position of the nodes.

Sort influence. This graph is one of the few cases where a sorting of the nodes is not advised, as it disturbs the recursive layout algorithm.

Pro et contra. The advantage of this graph is that it can render a complex system in a very simple manner. Its only drawback is that because the position of the nodes is defined by the layout algorithm, this graph tends to get very large for big systems and will sometimes not fit on one single screen. The use of node shrinking can alleviate this problem.

2.2.9.2 The Correlation Graph

Figure 2.7: A correlation graph of method nodes using LOC and NOS as position metrics.

Overall Idea. This graph can render the relationship between two metrics when they are applied to entities. The two metrics are directly mapped onto the position coordinates of the nodes. This graph needs an absolute origin within a coordinate system, which in our case is the upper left corner of the graph. If the chosen metrics are in close relation to each other, the nodes are positioned along a certain correlation axis, which is defined by the metrics. If a node finds itself far away from this correlation axis, it means that its metric measurements are somehow abnormal compared to the other nodes and should be inspected. Very large measurements put a node far away from the origin, if one of the two position metric measurements is very small, the node finds itself near the left or top border of the graph.

Scope. This graph can be applied to any type of entity. The maximum number of displayable nodes is very big, as the expansion of the graph drawing depends on the outliers in the system and not on the number of displayed nodes. This involves an overlapping of nodes, which however is not negative, because we are mainly interested in the outliers (i.e. the extreme values).

Layouts. There is only one possible layout in this case, which directly maps the position metrics to the position of the nodes.

Metrics. The number of possible metrics that can be applied is 5. Indeed, each metric can be applied in this case. However, if we choose to select size metrics this involves that the nodes overlap, while without size metrics the nodes will either be positioned next to each other or cover up other nodes entirely. The overlap problem is especially acute when the chosen size metrics tend to have big values, like LOC.

Sort influence. A sort has no influence on the layout.

Pro et contra. The main advantage of this graph is its scalability. Another advantage is that we can pick out the outliers at one glance. The drawback is a certain loss of overview, because the nodes overlap. However, as we often do not make use of size metrics for this graph, we can circumnavigate this problem.

2.2.9.3 The Histogram

Figure 2.8: A horizontal histogram.

Figure 2.9: A horizontal histogram using the size addition layout

Overall Idea. A histogram provides a representation of the distribution of entities related to a certain metric. The distribution of the nodes can in turn give us general information about a system. For example if we use as vertical position metric LOC of methods, we are able to gather if the methods tend to be overlong or not, and if there are any significant outliers.

Scope. This graph can be applied to any type of entity, class, method or attribute. The number of displayable nodes is also very large. However, since a large part of the nodes distribute around a certain value, a few of the rows of this graph can get very large and eventually get bigger than the screen. This problem is sometimes acute if we use the size addition layout described below. One of the fields where its use is advised, is to make a distribution of the methods of single classes or of attributes of subsystems.

Layouts. There are two possible layouts. The first, called *horizontal*, ignores size metrics and displays every node with the same size. The second one, called *size addition*, makes use of the width metric, and puts the nodes next to each other, while taking their size in consideration. Only the horizontal layout can be considered to be a real histogram, the kind which is used in the field of statistics.

Metrics. The number of possible metrics depends on the used layout. The horizontal layout can make use of 2 metrics, namely the color and the vertical position.The size addition layout can also make use of the width metric.

Sort influence. In the case of the horizontal layout, a sort has a positive effect if we take the color metric as sort criterion. It makes the detection of color metric outliers easier. In the case of the size addition layout, a sort according to the width metric also has some positive effect for the detection of width metric outliers.

Pro et contra. This graph shows a good behaviour in terms of scalability. Its major drawback is that the vertical position metric needs to have a rather large measurement interval, otherwise the nodes will be distributed all near the same vertical position.

2.2.9.4 The Checker Graph

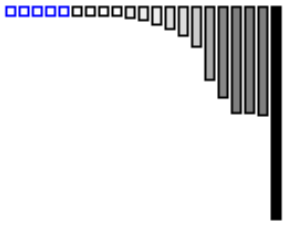


Figure 2.10: A checker graph using a sorted horizontal layout.

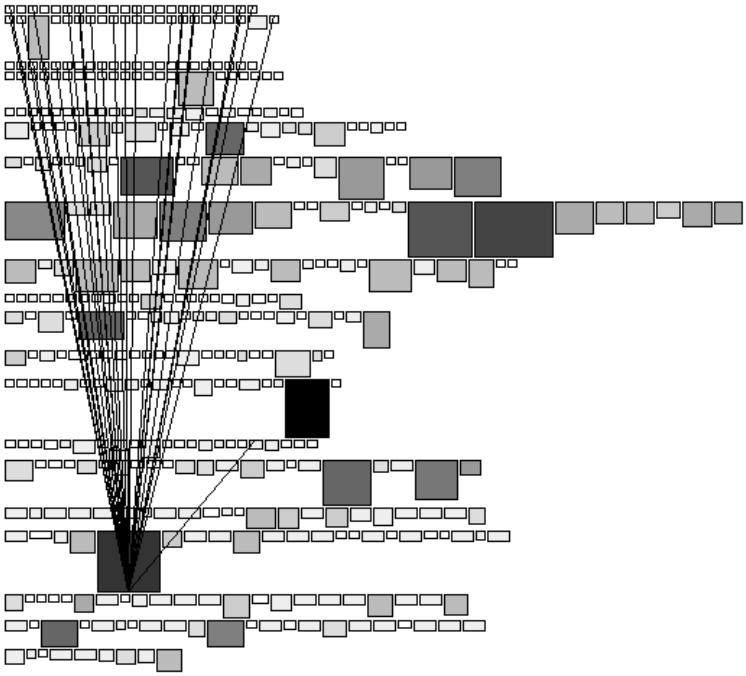


Figure 2.11: A checker graph using a quadratic layout with method nodes and invocation edges.

Overall Idea. The base idea for this kind of graph is simplicity. We want to lay out nodes without a special algorithm, we just place them one next to each other, to prevent them from overlapping.

Scope. This graph scales up quite well (especially if node shrinking is applied). Therefore it can be used for any kind on entity. However, it's not advisable to use edges in this graph, because it looks very chaotic, as they will cross the nodes.

Layouts. The first layout kind is called *horizontal* and *vertical*. We just place the nodes next to each other. We see such a layout in Figure 2.10[6]. Because this wastes a lot of space, we introduced the *quadratic* layout which tries to lay out the nodes to make them form a rectangle, whose width is dependent of the number of

[6]This figure suggests that a histogram is a special case of a checker graph. This is not true: a histogram makes use of a more complex layout algorithm which makes use of position metrics, as we see in the following subsections.

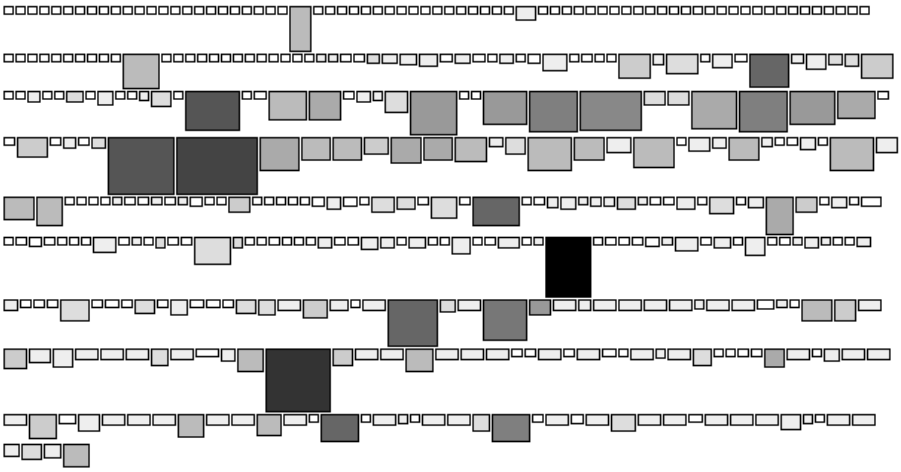


Figure 2.12: A checker graph using a maximal space usage layout.

displayed nodes. The graph which makes the best use of space is called *maximal space usage*, which tries to put as many nodes on the visible part of the drawing as possible.

Metrics. As the position metrics can't be used in this graph, we can only use size and color metrics.

Sort influence. The sort is essential for this graph. Indeed, if we don't make use of it, the nodes are placed randomly on the screen and it will be very hard to discern significant nodes. If we do make use of a sort according to a metric (especially the width metric), the detection of outliers will be very easy.

Pro et contra. The advantage is that we end up with a very easy to analyse layout. If the nodes are sorted, the detection of outliers is very easy, and the detection of suspicious node shapes is easy as well. This graph scales up well and several hundreds of nodes can be displayed at the same time without overlapping.

2.2.9.5 The Stapled Graph

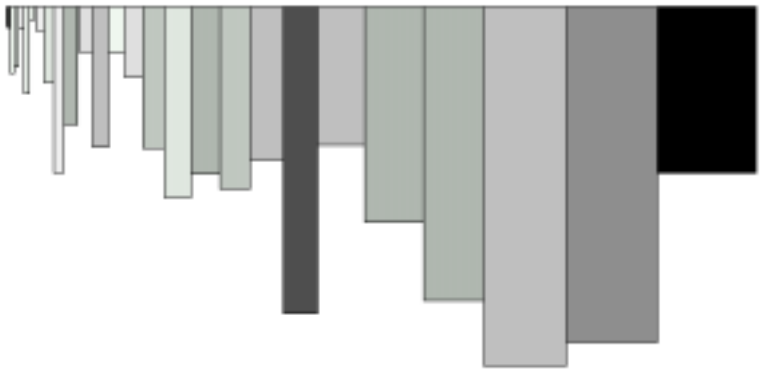


Figure 2.13: A stapled graph of class nodes.

Overall Idea. The idea for this graph came up when we tried to cure a small flaw in the horizontal checker layout: The width of the whole graph is defined by the summed widths of the nodes and cannot be influenced by the user. In such cases it often happens that the checker graph is wider than the screen. The stapled graph is thus a derivate: the user can indicate the maximum width of the graph he'd like to have, and all the node are accordingly shrunk in their width to make the graph fit the indicated space.

Scope. This graph can also display any kind of entity.

Layouts. A this time there is only one possible layout, which displays the nodes horizontally.

Metrics. The size and color metrics can be used, while this is not possible for the position metrics.

Sort influence. The sorting of nodes is essential for this graph to get some meaningful results. In fact it can be used for the detection of outliers regarding the height metric, if the nodes are sorted according to the width metric. If the two metrics are in close relation we often get a "staircase effect" because the nodes tend to get equally bigger in width and height. If this is not the case, the staircase effect breaks and we'll be able to easily detect those cases.

Pro et contra. One major drawback is that the width of a node will not directly reflect its metric, because it's being distorted by the graph width mapping function. Another drawback is that if the summed undistorted node widths of all nodes is bigger than the desired graph width, the nodes are shrunk in their width (otherwise they will be enlarged). If this shrinking is heavy, many small nodes will somehow disappear because they get very narrow, often only one pixel wide. The pro is obviously the intuitive detection of abnormal nodes which *don't* have to be outliers, but which stand out because two normally related metrics are not closely related in their case. Another pro is also that the graph will always fit the screen.

2.2.9.6 The Confrontation Graph

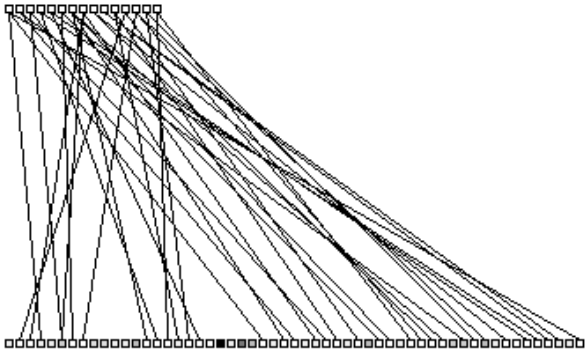


Figure 2.14: A confrontation graph using an horizontal layout

Overall Idea. This graph grew out of the necessity to display the access relationships between methods and attributes. An access is the only type of relationship between two entities of a different type.

Scope. This graph can only be applied on methods and attributes at the same time with accesses as edges. It's best used with the methods and attributes of one class.

Layouts. There are two possible displays. The first, called either *horizontal* or *vertical* displays on one row (column) the attributes and on the other one the methods. We can see such a layout in Figure 2.14. However, since in a class often the number of methods is much greater than the number of attributes, and the graph very soon gets larger than the screen, we introduced the *three row* layout. In this case the attributes are in the middle row, while the methods are in the upper and lower row.

Metrics. The size and color metrics can be used, while this is not possible for the position metrics.

Sort influence. A sort is advised for this graph. In the case of the method nodes it's especially useful according to the metrics LOC, NOS and NMAA. In case of the attribute nodes it's best to use NAA. If such a sort is applied, the number of edge crossings tends to drop and makes the graph look less cluttered.

Pro et contra. The major contra for this graph is that there is no special ordering of the nodes like clustering, except for a possible sort. However, it's the best graph to look at the internals of a class.

2.2.10 Useful Graphs: Class Graphs

In this section we list all graphs which display class nodes. We have noticed that the following graphs can be separated in two distinct groups. The graphs in the second group are normally applied after those in the first group, because they address more precise issues. We distinguish the following groups:

1. Those which serve primarily for system understanding. They work at a higher abstraction level, and in some cases can only return a general statement about the system. Problem detection is secondary in such graphs and in some cases not even possible. The following graphs fall under this category:
   - SYSTEM COMPLEXITY, Section 2.2.10.1.
   - SYSTEM HOT SPOTS, Section 2.2.10.2.
   - WEIGHT DISTRIBUTION, Section 2.2.10.3.
   - ROOT CLASS DETECTION, Section 2.2.10.4.
2. Those which primarily address problem detection, and secondarily program understanding. They must be applied on subsystems, rather than full systems. We list the following:
   - SERVICE CLASS DETECTION, Section 2.2.10.5.
   - COHESION OVERVIEW, Section 2.2.10.6.
   - SPINOFF HIERARCHY, Section 2.2.10.7.
   - INHERITANCE IMPACT, Section 2.2.10.8.
   - INTERMEDIATE ABSTRACT, Section 2.2.10.9.

2.2.10.1 System Complexity

Graph	Inheritance tree, without sort.	
Scope	Full system.	
Metrics		
Size	NIV (number of instance variables)	NOM (number of methods)
Color	WLOC (lines of code)	
Position	-	-

General Idea: This is one of the first graphs that should be applied to a system. It is an overview of the inheritance hierarchies of a whole system. This graph can give clues on the complexity and structure of the system (how many classes are present?), as well as information on the use of inheritance in the system (how deep do the hierarchies go and is the system in general flat or deep?). If we furthermore apply some class complexity metrics we can extract some more information. In this case we use as size metrics NIV and NOM, while for the color we choose WLOC. The detection of aberrant classes is now made easy: we can see if there are *very large classes*, *small classes* or even *empty classes*.

Results with the Refactoring Browser: In Figure 2.15 we see the SYSTEM COMPLEXITY graph applied on the Refactoring Browser. It shows few stand-alone classes and a few deep hierarchies. The first thing that strikes the eye is the class *BrowserNavigator* (A) which has a huge number of methods (175) and lines of code (1495) compared to the other classes present in the system. At the same time it only has one instance variable (this is the reason for its very narrow look). It may be a case for refactoring. If we take a look at the inheritance tree on the right side we can spot the class *BRStatementNode* (B) which is completely empty. When I asked the developers of the Refactoring Browser about this case, they told me that they were aware of the problem and that this class had been created to duplicate a hierarchy of another program. The same case can be spotted on one of the stand-alone classes *RefactoringError* (D) which is also empty. The next point of interest is the class *BRScanner* (C) which has the most instance variables (14) while it implements comparatively few methods (52). Perhaps this massive stand-alone class could be split up into subclasses. Another thing we can see is, that in the inheritance hierarchy in the middle of the graph, the root class *Refactoring* (E) is implementing by far the most methods, while there are quite a few very small classes deeper down the inheritance chain.

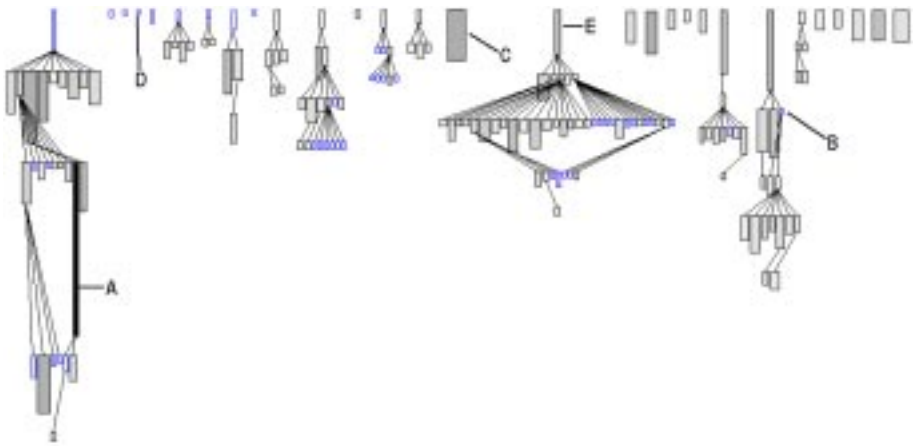


Figure 2.15: The system complexity graph applied on the Refactoring Browser using as size metrics NIV and NOM, and as color metric WLOC.

Results with Duploc: When we apply the SYSTEM COMPLEXITY graph on Duploc, we can spot the following in Figure 2.16: The system shows some very flat inheritance hierarchies, with many stand-alone classes which can have considerable sizes. This could mean that the system has not yet been refactored. There are three deep hierarchies, although in all three we can see that the main work is being done by the roots, which indicates top-heavy hierarchies. We also see that the main class *called DuplocApplication* (A) is very large and has only one very small subclass called it DuplocInformationMural[7]. Although *DuplocApplication* has the most methods and has the second most instance variables, the class with the most lines of code is *FastSparseCMatrix* (B). This class has only half the number of methods of *DuplocApplication* (74 vs. 130) but has nearly twice as much lines of code (1641 vs. 1060). Because of this we can already deduce that *FastSparseCMatrix* has some very long methods. The third point of interest are the classes on the left side (C): all of them are empty. These classes have become empty after being exported from the ENVY environment. The fourth eye-catch is the class *BinValueColoringModel* (D) on the right side. This class has the most instance variables (20), but only 52 methods. This may indicate that it is a service class which implements a lot of accessor methods. This supposition is being enforced by the light color value which is a sign for few lines of code (402), and is confirmed when we browse the source code of this class.

Figure 2.16: The system complexity graph applied on Duploc using as size metrics NIV and NOM, and as color metric WLOC.

Possible Alternatives: The color metric can be varied at will, especially class complexity metrics like NCV (number of class variables) prove to be useful.

Evaluation: This is certainly one of the first graphs that should be applied to a system, as it can return information on the structure and complexity of the subject system. However, it suffers one small drawback, which shows in very large systems: Sometimes the number of classes we want to display is so large that this graph takes several screens of place. It is difficult then to discern the outliers in the systems at one glance. The system hot spots graph discussed in Section 2.2.10.2 can counter this problem.

[7] The InformationMural is a subapplication of Duploc included in a latter phase of development. Evidently the developer did not want to write an own main application class from scratch, but preferred to take the existing one, subclass it and override only some needed methods. This explains the small size of this class.

2.2.10.2 System Hot spots

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system.	
Metrics		
Size	NOM (number of methods)	NIV (number of instance variables)
Color	WLOC (lines of code)	
Position	-	-

General Idea: For very large systems it's hard to decide where to start looking for problems hot spots. One general rule is to look for very large or complex classes regarding their number of attributes and methods. The graph described here is a very simple display of all classes in the system sorted according to a certain metric. The nodes are placed next to each other to prevent overlapping. This graph detects outliers very easily because of the sorting. We distinguish the following:

- Large nodes at the bottom of the graph. These represent the biggest classes in the system.
- Small nodes at the top of the graph. These are the smallest classes which can sometimes even be empty.
- Very flat nodes. These nodes possess very few (if any) instance variables.
- Rather high nodes. This is seldom the case, as classes rarely have many attributes. Sometimes we can detect configuration classes like this.

Results with the Refactoring Browser: In Figure 2.17 we get a HOT SPOTS view on the Refactoring Browser. While in Figure 2.15 we had to search for the biggest and smallest nodes, this is made easy in this kind of graph because the nodes have been sorted: as before we can locate the class *BrowserNavigator* (A) and *BRScanner* (B). The sorting of the nodes makes it easy now to detect empty or very small classes, which find themselves at the top of the graph (D). Our attention is now also drawn to other classes like *BrowserApplicationModel* (C), which implements 38 methods while it defines no instance variable, which is visible by its flat shape. The view on the shape of the nodes is also facilitated, we can now detect classes like *MoveVariableDefinitionRefactoring* (E), which defines 6 instance variables while it implements only 7 methods (mainly accessors), giving it nearly a square shape.

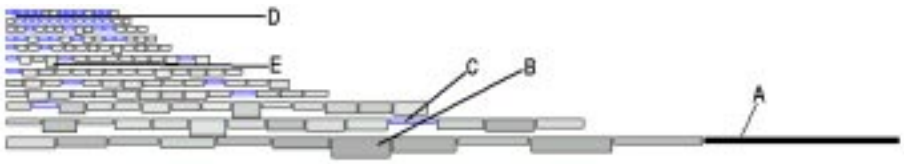


Figure 2.17: The system hot spots graph applied on the Refactoring Browser using as size metrics NOM and NIV, and as color metric WLOC. The nodes have been sorted according to NOM.

Results with Duploc: The HOT POTS view on Duploc reveals also some information which could not be seen at first sight in Figure 2.16, as we see in Figure 2.18. We see Duploc has either very large classes (A)(B), or very small ones (D). We can also locate some classes with many instance variables (C). Two classes which could be interesting for further investigation because of are *DuplocCodeReader* (F) (32 methods, 17 instance variables) and *DuplocProgressMeter* (E) (15 methods, 9 instance variables): both classes have many instance variables and few methods, which could indicate service classes apt for refactoring.

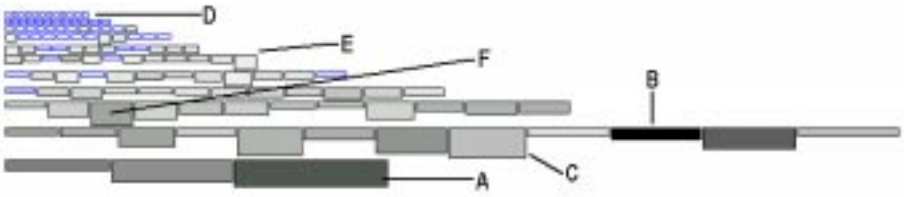


Figure 2.18: The system hot spots graph applied on Duploc using as size metrics NOM and NIV, and as color metric WLOC. Sort according to NOM.

Possible Alternatives: The color metric can be varied at will. A sort according to other metrics (especially WLOC and NCV) can also give interesting results which emphasise certain nodes.

Evaluation: The main drawback of the SYSTEM COMPLEXITY graph described in Section 2.2.10.1 is the fact that through the ordering of the nodes in tree structures we lose track of the size of the nodes all too easily. Only extreme cases strike our eyes. The SYSTEM HOT SPOTS graph described here makes this up through the sorting of the nodes and an ordering of them which reflects this sorting. However we lose the notion of inheritance in this case, since displaying the edges would mess up the view. A certain disadvantage of this graph is that the more nodes we display the more space is needed.

2.2.10.3 Weight Distribution

Graph	Histogram, size addition layout, sort according to width metric.	
Scope	Full system.	
Metrics		
Size	NOM (number of methods)	-
Color	HNL (hierarchy nesting level)	
Position	-	NOM

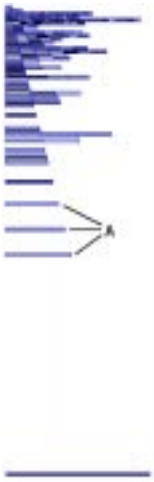


Figure 2.19: The weight distribution graph applied on the Refactoring Browser. As width and vertical position metric we use NOM, as color metric we use HNL.

General Idea: With this graph we are able to make a general assessment on the system we are investigating. The width and the vertical position of the nodes is reflected by NOM, the color represents their HNL. This means that the deeper down (in the graph) the class nodes are, the more methods these classes implement. A dark node on the other hand means that the class it represents has a deep hierarchy nesting level. The possible assessments we can now make are:

- The system is *top-heavy*. This means that the classes that implement the most functionality are high up in the inheritance hierarchies. Such a graph has big nodes (on the bottom of the graph) which have very light color values (because their HNL is small). Top-heavy systems suffer when it comes to subclassing and reusing, because their root classes do too much themselves.
- The system is *bottom-heavy*. The most functionality is implemented in classes deep down the inheritance hierarchies. Such a case displays dark, big nodes on the bottom of the graph. Bottom-heavy systems are sometimes the results of overzealous abstracting mechanisms.
- The system is *even*. This display looks somehow chaotic, because the dark and light nodes distribute themselves over the whole graph. This case balances the two cases described above.

Results with the Refactoring Browser: The Refactoring Browser is an evenly distributed system, as we see in Figure 2.19: It's not possible to locate a majority of the dark or the light nodes on a certain area of the graph, although we can see there are three big classes marked as (A) high up the hierarchy.

Figure 2.20: The weight distribution graph applied on Duploc. As width and vertical position metric we use NOM, as color metric we use HNL.

Results with Duploc: Duploc is clearly a top-heavy system, as we see in Figure 2.20: The dark nodes are all very small (small NOM) and thus located on the top region of the graph. The big classes on the bottom of the graph are all very light (high up in the hierarchy). The system is thus to be classified as top-heavy, which is mainly due to its young age: Duploc has not yet undergone a reengineering or refactoring. It should be analysed on whether it's possible to introduce a supplemental abstraction level high up in the hierarchy.

Possible Alternatives: The width metric can be varied, especially NIV (number of instance variables) can give some supplemental information on the complexity of the classes. The color metric can also be changed, especially WLOC (lines of code) shows a good behaviour.

Evaluation: This graph can make a general assessment about the system. Such an assessment may not be very useful and will most probably not involve a specific problem, but upon such statements about the subject system we can vary our approach. In fact, the more we know about the system before we dive into its details, the more precisely we can deploy the other graphs.

2.2.10.4 Root Class Detection

Graph	Correlation.	
Scope	Full system or very large subsystems.	
Metrics		
Size	*	*
Color	*	
Position	WNOC (total number of children)	NOC (number of children)

General Idea: In very large systems with many inheritance hierarchies it may be difficult to identify at once the classes which have the most impact on their subclasses. The impact of a class on its descendants can be measured with the number of direct subclasses and the total number of subclasses of a class: the more there are, the more the functionality implemented in a root class is used. This graph shows the correlation between WNOC (total number of subclasses) and NOC (number of direct subclasses).

The further away from the origin such a class node is, the bigger is its impact. The type of inheritance used for a class can also be identified with this graph:

- If a node is positioned on the right side of the graph, while holding a vertical position near the top, this means that while the underlying class has a great number of descendants its direct subclasses are few. This is often the case when directly below a root class a supplemental abstraction level of classes has been introduced.
- If the node finds itself on the 45 degrees axis (it can't be further left because WNOC is always at least equal to NOC) and far away from the top of the graph, this means that the underlying class has a lot of direct subclasses. This is what we call a *flying saucer hierarchy* because the inheritance tree of this class resembles one.
- If a class node is positioned exactly along the 45 degrees axis this means that all its subclasses don't have subclasses themselves, and thus are leaf node classes in an inheritance tree.

Results with the Refactoring Browser: To make the effect of this graph more visible, in Figure 2.21 we see on the top left the root class detection graph while on the bottom right we see a display of two major inheritance trees. We see the class *Refactoring* (A) which has 43 descendants and 5 direct subclasses as root of major inheritance tree on the right side of the correlation graph. The other root class, *BrowserApplicationModel* (B) can also be identified on the right side of the graph. Two classes, *MethodRefactoring* (C) and *VariableRefactoring* (D), which are the heads of minor flying saucer hierarchies (14 and 13 direct subclasses) can be identified near the 45 degrees axis.

Results with Duploc: The results of this graph are somewhat deceiving in the case of Duploc, as its inheritance hierarchies are very flat. We can detect however two root classes, namely *PresentationModel-ControllerState* (A) and *PMCS* (B). In Figure 2.22 we see where the detected root classes are located in one of the inheritance hierarchies of Duploc.

Possible Alternatives: We do not make use of the color and size metrics, which could add information to this graph.

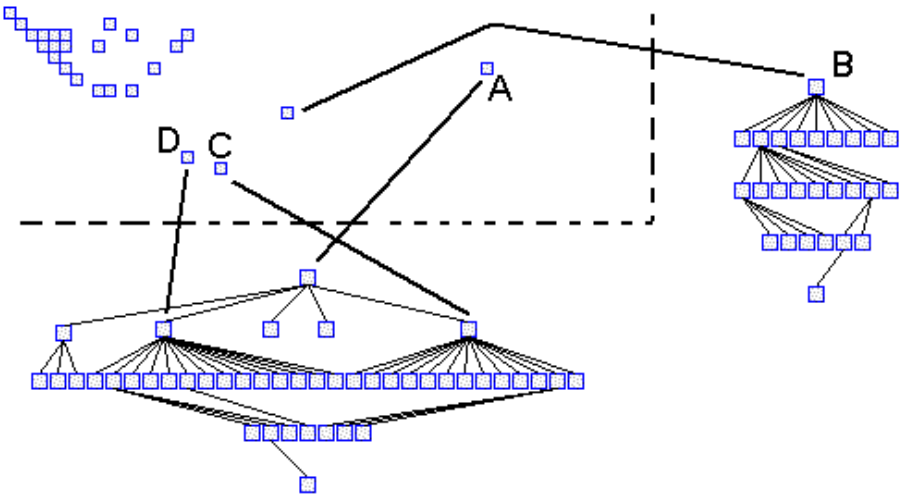


Figure 2.21: A root class detection graph applied on the Refactoring Browser. As position metrics we use WNOC and NOC.

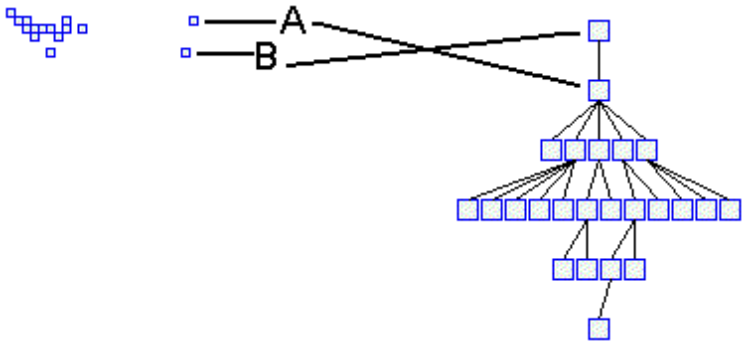


Figure 2.22: A root class detection graph applied on Duploc. As position metrics we use WNOC and NOC.

Evaluation: The detection of flying saucer hierarchies can of course be done through an inheritance tree display. The resulting tree graph has then to be searched for them. However, in some cases where the number of classes was very large, the resulting graph would become several screens big. In such cases it's not easy to detect flying saucers at once, and the graph described here comes into play. This graph can come in handy to see if there are some inheritance hierarchies upon which we want to apply inheritance specific graphs like intermediate abstract or inheritance impact.

2.2.10.5 Service Class Detection

Graph	Stapled, sort according to width metric.	
Scope	Subsystem or small full system.	
Metrics		
Size	NOM (number of methods)	WLOC (lines of code)
Color	NOM	
Position	-	-

General Idea: This graph has proven to be useful for the detection of so-called *service classes*. A service class is a class which mainly provides services to other classes. It often contains some tables and dictionaries which other classes can access for their purposes. Such classes often have an aberrant ratio between NOM and WLOC: they have very short methods which mainly access or return values. In this kind of graph we present a selection of some classes (a whole inheritance tree is often a good choice) as a stapled graph. The classes have been sorted according to their width, which represents NOM.

Because there tends to be a certain relation between NOM and WLOC, we should get a sort of staircase effect on the nodes the more we move to the right.

We can make out the following:

- If a class node breaks the staircase effect (by being too short) it is a candidate for a service class.
- This graph can also serve as detector for classes with overlong methods: If the class breaks the effect in the other direction (by being too tall) it's a candidate for method splitting, because this means that it has many lines of code (tall) and comparatively few methods (narrow, and because of the sorting pushed to the left side of the graph).

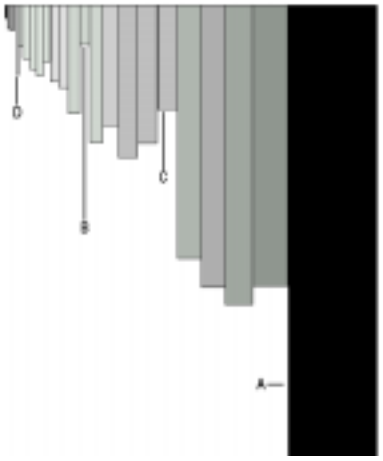


Figure 2.23: The service class detection graph applied on a subhierarchy of the Refactoring Browser. As width metric and sorting criterion we use NOM, the height metric is WLOC.

Results with the Refactoring Browser: In Figure 2.23 we selected a whole inheritance tree (26 classes) of the Refactoring Browser to be displayed in a SERVICE CLASS DETECTION graph. We see one huge class *BrowserNavigator* (A), which in fact is even bigger, but we cut it down because of space reasons. We see quite clearly that there is a certain tendency for a staircase which is severely broken in two places. The first

service class candidate is *CodeTool* (B), which has 22 methods and 49 lines of code. A closer inspection reveals that the methods are mainly get/set-methods (accessors). The second candidate is *CodeModel* (C) with 40 methods and 136 lines of code. The name itself already reveals the service function this class is intended to have. As method splitting candidate we detect the class *ClassCommentTool* (D) which has only 7 methods but 89 lines of code.

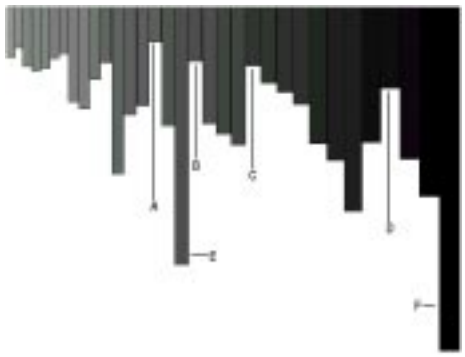


Figure 2.24: The service class detection graph applied on a subset of Duploc. As width metric and sorting criterion we use NOM, the height metric is WLOC.

Results with Duploc: We obtained the graph in Figure 2.24 by first applying the graph on the whole system and then by selecting a subset which looked interesting. We see there are some candidates for service classes: The class *CachedObservationData* (A) contains 20 methods for a total count of 50 lines of code. A closer inspection reveals it is truly a service class. Nearly the same ratio is visible in the classes *ComparisonMatrixBody* (B) (22/80), *PresentationModelControllerState* (C) (25/87) and *ObservationOnRawSubMatrix* (D) (30/122). Some classes tend to have overlong methods, namely *PMVSInformationMuralMode* (E) (22/396) and *DuplocCodeReader* (F) (32/530), and should be looked at for possible method splitting.

Possible Alternatives: Nearly the same results can be obtained if we use NIV (number of instance variables) instead of NOM: both NOM and NIV are closely related in service classes (because of the accessors). Sometimes abstract classes higher up the hierarchy tend to have the same properties as service classes, because their abstract nature makes them have several very short methods which are later overridden or extended by the subclasses. This can be alleviated if we use HNL (hierarchy nesting level) as color metric. Service class candidates which are true service classes tend then to have a darker color shade. Fake service classes like the abstract ones will have a lighter color shade because they are higher up the hierarchy.

Evaluation: As this graph addresses a special problem, it should be used in a second phase of reverse engineering. Experience has shown that it's advisable to apply it on subsystems, especially inheritance hierarchies.

2.2.10.6 Cohesion Overview

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem.	
Metrics		
Size	NOM (number of methods)	WNAA (number of direct accesses on attributes)
Color	NIV (number of instance variables)	
Position	-	-

General Idea: In this graph the nodes differ greatly in shape and color. In the best case this graph can give us some clues on which classes we should inspect for a possible splitting. We distinguish the following:

- The flat nodes indicate that the methods of a class (the width indicates the number) do not access many times its instance variables. This is further emphasised by the small height (few instance variable accesses).
- The narrow and high nodes on the other hand, tend also to be very light colored. This case happens when the classes have many accesses but only few instance variables. This is mostly the case when the class defines an attribute which is then heavily accessed directly by its subclasses. This is not advisable because of the lacking encapsulation: a single access through an accessor which would then be invoked by other classes, instead of direct accesses on the attribute, would be much better.
- More or less rectangular nodes with darker color shades indicate a good cohesion inside those classes, although this is only provable after applying a class cohesion graph, which is described in Section 2.2.13.1.

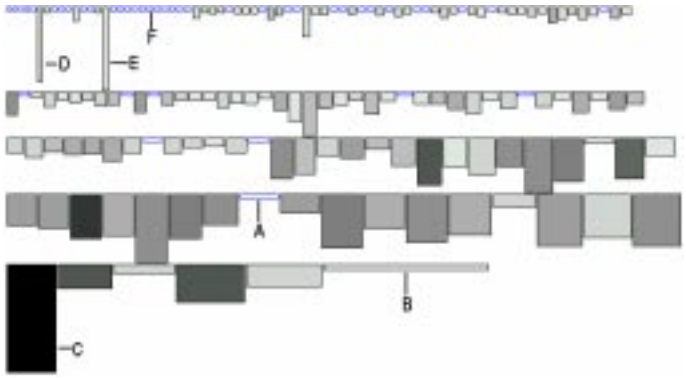


Figure 2.25: A cohesion overview graph applied on the Refactoring Browser. As size metrics we use NOM and WNAA. As color metric NIV is used.

Results with the Refactoring Browser: The resulting graph can be seen in Figure 2.25. The first thing we notice is that the nodes differ heavily in their shapes and colors. There are some white nodes that don't define instance variables (for example (A)) and because of this absence they can't have any instance variable access either. This is the reason for their flat shape. We also gather there are some empty or nearly empty ones (located around (F)). The class *BrowserNavigator* strikes once again the eye for its huge number of methods and its small number of instance variables (only one). The nodes (D) and (E) strike the

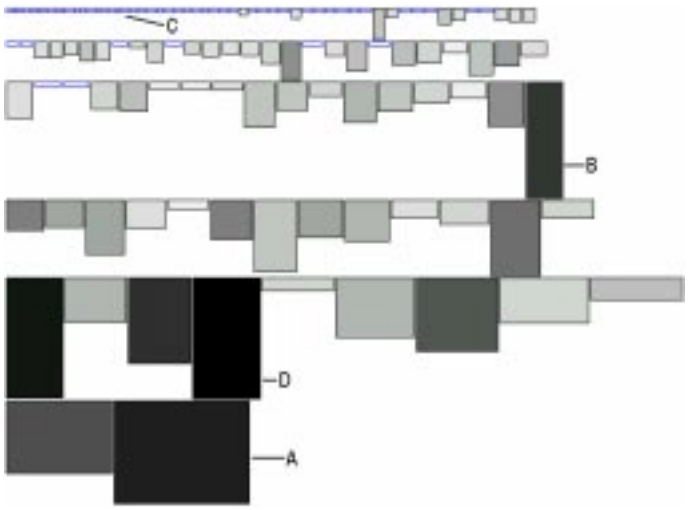


Figure 2.26: A cohesion overview graph applied on Duploc. As size metrics we use NOM and WNAA. As color metric NIV is used.

eye for their narrow shape and light color: Both have few methods and instance variables, (1,2) and (2,1) respectively, while at the same time they have a huge number of accesses. The reason for this is that their variables are directly accessed by their subclasses. The class *BRScanner* (C) shows a great complexity and heavy access.

Results with Duploc: The graph in Figure 2.26 shows a few characteristics of Duploc: Many empty or nearly-empty classes (C), quite a few heavy-access classes (B) and (D) and a few very large classes, for example *DuplocApplication* (A). We see there are quite a few classes that could be interesting for inspection with a class cohesion graph and do that for one special case, the class *DuplocApplication* in Section 2.2.13.1.

Possible Alternatives: None.

Evaluation: This graph can be seen as an *in-betweener*, because it comes after a graph for general overview and before a graph which treats class internals. The best result it can return is a collection of classes which we should further examine with a class cohesion graph, described in Section 2.2.13.1.

2.2.10.7 Spinoff Hierarchy

Graph	Inheritance tree, centered, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	WNOC (total number of children)	NOM (number of methods)
Color	WNOC (total number of children)	
Position	-	-

General Idea: We have noticed that in inheritance hierarchies the notion of inheritance is often carried on only by one or two classes on each level of the inheritance tree. This means that when a class has some subclasses often only one of them is really carrying on the weight of the inheritance, while its siblings tend to be *spinoff* classes implementing only few functionalities. Although this is not a bad thing per se, an easy detection of such spinoff hierarchies could make us focus on the inheritance carriers, while we could save time by ignoring (at least at the beginning) the less important spinoff classes. Spinoff classes often implement few methods and have few or no subclasses at all.

We distinguish the following:

- Small, light colored nodes. These are the *spinoff classes* with few or no children and few methods.
- Large, dark colored nodes. These are the *inheritance carriers*.

Results with the Refactoring Browser: In Figure 2.27 we see all inheritance hierarchies that make up the Refactoring Browser. We filtered out all stand-alone classes to get a clearer overview. We detect two cases of spinoff hierarchies:

1. The one with the class *BrowserApplicationModel* (A) as root. We see two classes split up the second level of this tree, namely *CodeTool* (A21) and *Navigator* (A11). There are a few spinoff classes on this level, neither of them has subclasses. The same situation is present on the next level of this tree where the classes *BrowserTextTool* (A22) and *BrowserNavigator* (A12) carry on the weight of inheritance. A good example for spinoffs is visible between *CodeTool* (A21) and *BrowserTextTool* (A22): *CodeTool* has 7 subclasses but only one of them, *BrowserTextTool*, carries on the inheritance. Each one of its siblings is very small (keep in mind that the height reflects NOM) and is thus a spinoff.
2. The one with the class *Refactoring* (B) as root. Again two main inheritance threads are visible: The one consisting of *Refactoring* (B), *MethodRefactoring* (B11) and *ChangeMethodNameRefactoring* (B12). The other consists of *Refactoring* (B), *VariableRefactoring* (B21) and *RestoringVariableRefactoring* (B22).

The other inheritance trees in this display also show some property of a spinoff hierarchy, and could be a case of further investigation.

Results with Duploc: After removing the many stand-alone classes from Duploc, the remaining graph in Figure 2.28 can only show us the absence of spinoff hierarchies. Especially in the tree with the class *PresentationModelControllerState* (A) as root, we see that on the third level we have 5 siblings, 4 of which are all inheritance carriers, with only one tiny spinoff class with the meaningful name *PMCSDummyMode* (B).

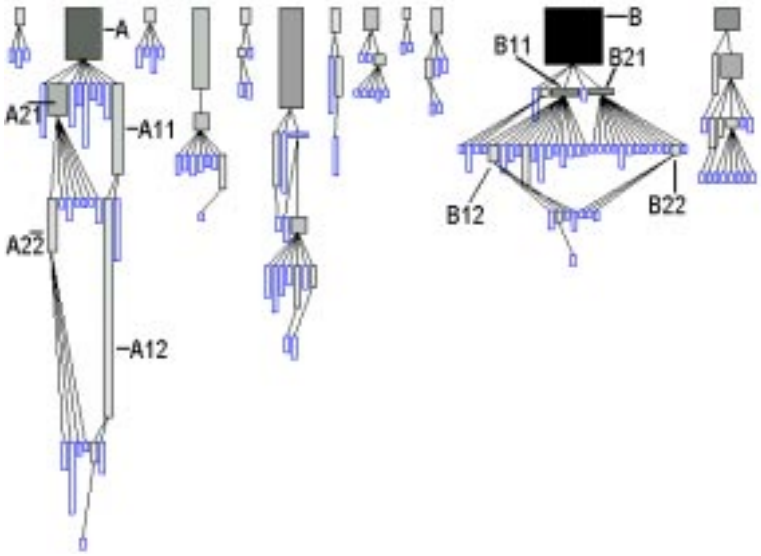


Figure 2.27: The spinoff hierarchy graph applied on the inheritance hierarchies of the Refactoring Browser. As size metrics we use WNOC and NOM, as color metric WNOC.

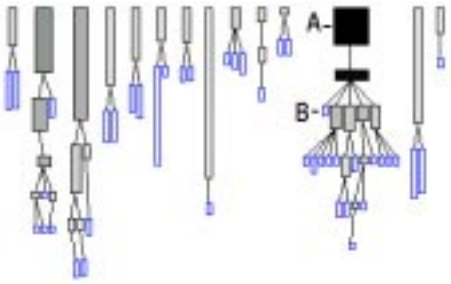


Figure 2.28: The spinoff hierarchy graph applied on Duploc. As size metrics we use WNOC and NOM, as color metric WNOC.

Possible Alternatives: We have to emphasise that a preprocessing consisting of filtering out all stand-alone nodes is advised for this graph, as they add unnecessary complexity to the displayed graph. This graph does not have real alternatives, as it addresses a special problem.

Evaluation: This graph should come into play in a later phase of the reverse engineering, as it addresses a special problem which may not be present at all in the system. The detection of an inheritance carrier could be important, as it is the place which should be checked out because subclasses depend on it. The spinoff classes on the other hand, can be examined for possible push-ups of functionality.

2.2.10.8 Inheritance Impact

Graph	Inheritance tree, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	NMO (number of methods overridden)	NME (number of methods extended)
Color	NOM (number of methods)	
Position	-	-

General Idea: This graph is able to tell us if there has been made an improper or suspect use of inheritance: it can tell us if a class that implements many methods does not make use of method overriding or method extension, or uses it only rarely. Overriding and extending methods is one of the powerful properties of object-oriented languages and should be used if possible.

Nodes that override or extend a lot are bigger, nodes that implement many methods are dark. We are looking for dark nodes (many methods) which are at the same time very small (no use or rare use of overriding and extension).

Results with the Refactoring Browser: One of the hierarchies of the Refactoring Browser seems to have one such class which should certainly be further investigated: In Figure 2.29 we can detect the class *BrowserNavigator* (A) which implements many methods (175), while it only overrides one and extends two methods.

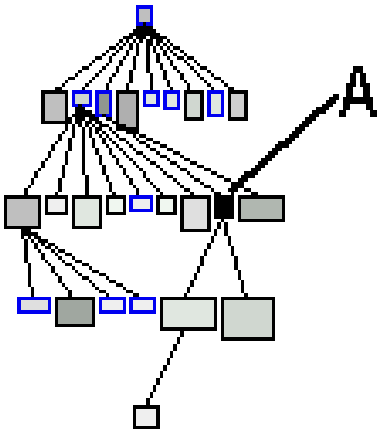


Figure 2.29: The inheritance impact graph applied on an inheritance tree of the Refactoring Browser. As size metrics we use NMO and NME, as color metric NOM.

Results with Duploc: This graph returns no meaningful results if applied on Duploc.

Possible Alternatives: No real alternatives, as it addresses a specific problem. This graph is often obtained after filtering out all stand-alone classes and all inheritance hierarchies which show no sign we are looking for.

Evaluation: A graph which addresses a very special problem. It's not always useful, but if it can detect something, it can be an important discovery which can affect a whole inheritance hierarchy.

2.2.10.9 Intermediate Abstract Class

Graph	Inheritance tree, without sort.	
Scope	Subsystem, especially inheritance hierarchies.	
Metrics		
Size	NOM (number of methods)	NMA (number of methods added)
Color	NOC (number of children)	
Position	-	-

General Idea: This graph is useful for the detection of abstract classes or nearly-empty classes which are located somewhere in the middle of an inheritance chain. Often they tend to have a superclass which implements a lot of methods. The programmer then started to subclass this class. The number of direct subclasses would soon be too big so an attempt was made to logically group several subclasses under an abstract intermediate class.

Such an intermediate subclass would normally have many children, while at the same time its size is very small (because it is abstract or nearly empty). We thus have to look for small, dark nodes in the middle of inheritance hierarchies.

The dark color comes from the greater number of direct subclasses, while the small size from the small functionality implemented. We chose NMA as height metric to reflect the fact that often such intermediate abstract classes don't override superclass methods, which in turn means that is we use NOM as width metric, the node is square (no functionality implemented, or if there is a bit of implemented functionality, then it doesn't come from the superclass). Intermediate abstract classes are of some interest, because often we can try to push up some functionalities of its subclasses into it, thus concentrating them in one place, instead of spreading the functionality all over the subclasses, risking to obtain duplicated code.

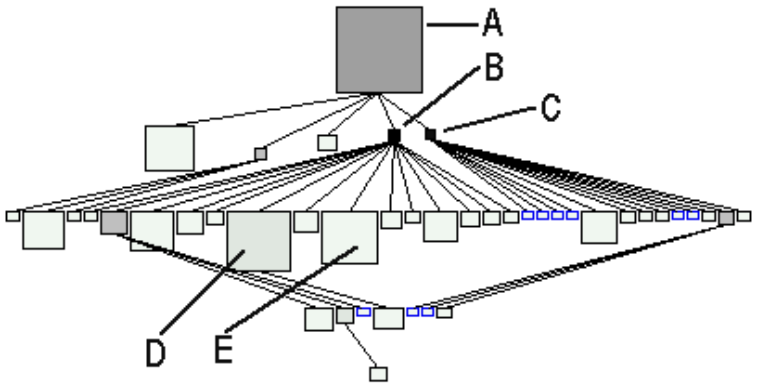


Figure 2.30: The intermediate abstract class graph applied on a subset of the Refactoring Browser. As size metrics we use NOM and NMA, as color metric NOC.

Results with the Refactoring Browser: The Refactoring Browser harbours in one of its inheritance hierarchies two intermediate abstract classes, as we see in Figure 2.30. The root class *Refactoring* (A) implements quite a few methods, while we can spot the two intermediate abstract classes as *MethodRefactoring* (B) and *VariableRefactoring* (C). These two classes implement themselves very few methods (2 and 1 respectively) and are the roots of smaller subhierarchies. In the case of *MethodRefactoring* we see that its

subclasses are implementing several methods, as we see in *InlineMethodRefactoring* (D) and *MoveMethodRefactoring* (E). Perhaps an attempt could be made to extract duplicated code and push it up into the intermediate abstract class.

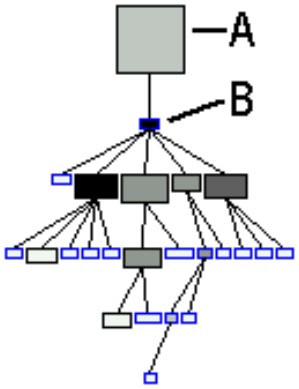


Figure 2.31: The intermediate abstract class graph applied on an inheritance hierarchy of Duploc. As size metrics we use NOM and NMA, as color metric NOC.

Results with Duploc: One of Duploc's inheritance hierarchies also contains an intermediate abstract class, as we see in Figure 2.31: The subclass *PMCS* (B) of the root class *PresentationModelControllerState* (A) implements only 4 methods and is obviously an intermediate abstract class. The subclasses of *PMCS* should be searched for duplicated code which could be pushed up into *PMCS*.

Possible Alternatives: None.

Evaluation: The detection of abstract classes is very important: several object oriented languages either directly provide a declaration or support a standard idiom for identifying abstract classes. Abstract or nearly abstract classes can be seen as the hinges of the system, upon which several classes depend. It's where the common functionality is defined and where we should start to look at source code if we want to understand the logic of their subclasses.

2.2.11 Useful Graphs: Method Graphs

Method graphs can work at any level of granularity most of the time. However, the more method nodes we display, the harder it is to make out outliers. Methods are the entities which are responsible for the action in a system. This implies that every graph which uses method nodes is often followed by an examination of the actual underlying source code. This means that the graphs listed here have a very concrete context.

In this section we list the following graphs:

- METHOD EFFICIENCY CORRELATION, Section 2.2.11.1.
- CODING IMPACT HISTOGRAM, Section 2.2.11.2.
- METHOD SIZE NESTING LEVEL, Section 2.2.11.3.

2.2.11.1 Method Efficiency Correlation

Graph	Correlation.	
Scope	Full system, subsystem or single class.	
Metrics		
Size	NOP (number of parameters)	NOP
Color	*	
Position	LOC (lines of code)	NOS (number of statements)

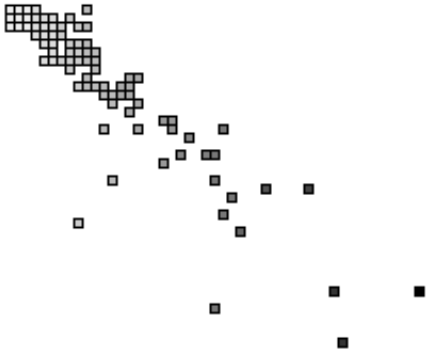


Figure 2.32: A method efficiency correlation graph.

General Idea: This graph is a good way to locate the *freaky entities* inside a group of methods, when it comes to their efficiency. By efficiency we mean how many statements are put on each line. By displaying the nodes in the correlation graph (as in Figure 2.32), we see that most of the nodes are near a certain correlation axis. However, there are a few which do not adhere to this rule.

The methods that are not near the correlation axis may have some problems, which may be

1. High LOC (lines of code) and low NOS (number of statements). This is for example the case with ”forgotten methods”, that at some point have been commented out and then been forgotten. This may also be the case for overzealous line indentation, when a single parenthesis is put on a line of its own or when many blank lines have been used.
2. Low LOC and high NOS. This can be the case when the methods are written without indentation and several statements are on the same line, which is a bad thing too, since this decreases the readability, and it may also break the law of Demeter [LIEB 89].
3. Long methods (high LOC and high NOS). Normally a case for redesign, since long methods should be split up in smaller, better understandable and reusable ones [BECK 97].
4. Empty methods. These nodes position themselves on the top left of the graph. Although they can be viewed there by selecting and moving, the overlapping of the nodes which is characteristic for this graph makes it hard to see those empty methods at one glance. A better graph for the detection of empty methods is the Coding Impact Histogram described in Section 2.2.11.2.

Other hot spots can be detected by looking at the size of the nodes:

- Big nodes have many parameters. Although it’s hard to define a threshold on the number of parameters, we think that methods taking more than 5 parameters should be looked at.

- Very small nodes on the outskirts of the graph should be looked at: these are very long methods which do not take any input parameter. Perhaps they could be split up easily.

The interesting property of this graph is its scalability. Since most of the nodes overlay each other, and those nodes are of no real interest to us, because they have average metric measurements, we can display several thousands of nodes at the same time. Our interest is drawn by the nodes which find themselves on the outer skirts of the graph, and which do not suffer overlaying, as their position is defined by their non-average metric measurements. The size of this graph is not affected by the number of displayed nodes, but on the maximum values for the position metrics.

Results with the Refactoring Browser: The method efficiency correlation graph shows some interesting results when applied to the Refactoring Browser. In Figure 2.33 we display all 2365 methods of the Refactoring Browser. We can spot several cases which should be looked into. The first nodes to meet the eye are those on the right edge of the graph (A). These three methods are very long (45, 51 and 65 lines of code) compared to the other methods in the system, which does not have a great distribution, thus signifying that the system is homogeneous related to the method lengths. The opposite case can be seen on the top left side of the graph (B). Upon closer inspection (by selecting and moving the nodes) we can see that the RefactoringBrowser contains 20 empty methods. The next point of interest is the method marked (C): this method takes 7 input parameters which is of course very much. The method *reInstallInterface* (D) on the top of the graph is also a case of closer study: While it has 16 lines of code it contains no statements. If we browse its source code, we see that the whole body of the method has been commented. The method *needsParenthesisFor:* (E) on the other hand contains 31 statements in only 19 lines of code and should perhaps be reformatted. The group of methods marked as (F) should also be looked into, since all of them contain comparatively few statements in long method bodies.

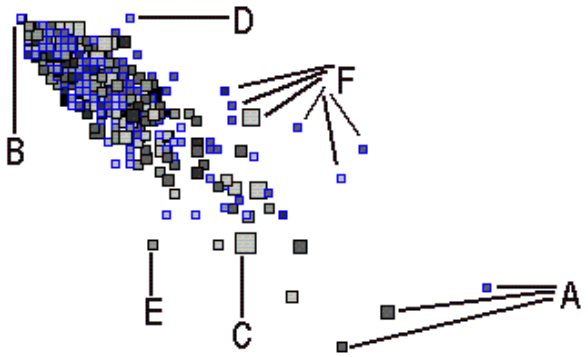


Figure 2.33: The method efficiency graph applied on the Refactoring Browser, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

Results with Duploc: When this graph is applied to Duploc, as we see in Figure 2.34, the first thing to strike the eye is the large distribution of the nodes. Duploc obviously does have some very long methods. The second thing that meets the eye is that the main correlation axis has a different angle compared to the Refactoring Browser in Figure 2.33. The method *putPerlCode:* (A) is 201 lines long but does have only 2 statements. Upon closer inspection we see that its purpose is to print out a very long string. We have some other very long methods, (B) with 135 lines, (C) with 95 lines, (D) with 109 lines. We have some method that are far away from the system correlation axis, like (A), (C), (E), (F) and (G). (E) for example has 64 lines of code with only 1 statement. A closer inspection reveals its body is mainly commented code for testing purposes, i.e. when the system is tested some parts of the method body are uncommented. (F) reveals the same situation, where the 18 lines long method body doesn't contain any statements at all. (G) has 32 statements packed in 14 lines of code. Reformatting makes it more readable. The empty methods

can of course be detected as (H), while we should also note the nodes around (I), which seem to be very short and at the same time badly formatted methods. The two methods (J) also draw attention due to their considerable size, which reflects the fact that they take 9 input parameters each.

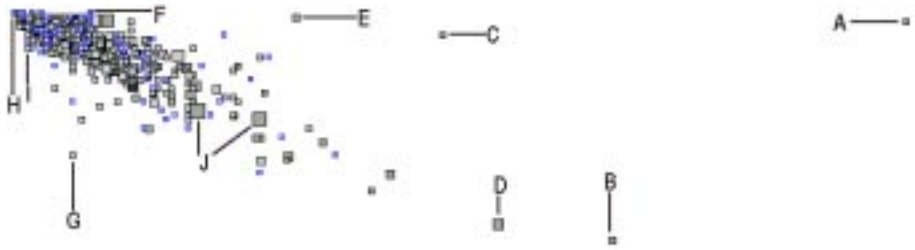


Figure 2.34: The method efficiency graph applied on Duploc, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

Possible Alternatives: We chose the size of the nodes to be represented by NOP (number of parameters). Since the distribution tends to get sparse the more we move to the right and to the bottom, we can see the methods which take many parameters more clearly, since it's normally the large methods that take more parameters. Generally in this graph the size metrics can be chosen freely, although it's advisable to use metrics which tend to have small measurements. Otherwise the nodes get very big and clutter up the view. The color metric can also be used freely. We chose HNL (hierarchy nesting level) in this case, but since the nodes in this graph tend to be very small, the color node metric doesn't really matter.

Evaluation: This is one of the few graphs which works very well at any level of granularity. As such it can be used anytime. We saw it can be useful to apply it on a subsystem before we dive into its details. At class level it can help to detect problem cases for a concrete reengineering.

2.2.11.2 Coding Impact Histogram

Graph	Histogram, size addition layout, sort according to width metric.	
Scope	Single class or small subsystem.	
Metrics		
Size	LOC (lines of code)	-
Color	LOC	
Position	LOC	-

General Idea: This graph shows the coding impact of methods and where the most coding has happened. While the normal histogram can only tell us how methods are distributed regarding their lines of code, this graph (Figure 2.35) can reveal where the real programming effort has been made: Writing 20 methods each one line long is easier than writing one method 20 lines of code long. It shows if there are any aberrant methods that are too long or if the system is unbalanced because of too long and complex methods. As a nice side-effect we can also grasp at one glance if there are any empty methods (those at the very top of the graph). A good design should have a lot of tiny methods so this is where the biggest columns in the graph should be. Methods not following this rule should be analysed as possible ”split candidates” which could be broken down into smaller pieces. While this graph is inefficient on whole systems because of the huge number of methods, it has proven to be very useful when applied to the methods of one single class. It should also be noted that the average length of a method implemented in typical industrial Smalltalk applications is around 6 lines [BECK 97].

Figure 2.35: A coding impact histogram.

Results with the Refactoring Browser: Since this is one of the graphs which can hardly be applied on whole systems, but rather on specific small subsystems or singular classes, we do not compare the systems from our case studies with each other, but we rather show a few illustrative examples taken out randomly[8] from the Refactoring Browser. We selected only the two classes (*BrowserNavigator* (B) and *BRProgramNode* (A)) with the most methods for this graph. We see in Figure 2.36 that each class has its own coding impact topography. We see that *BrowserNavigator* (B) has many methods which tend to be overlong, and especially 6 very long ones which isolate themselves (B1) from the others. On the other hand *BRProgramNode* has an irregular topography with many accessors (A2) and one very long method (A1).

Possible Alternatives: This graph knows many useful mutations, especially those which keep LOC as vertical position metric, but use other size and color metrics and a different sort criterion. In these cases, especially NI (number of invocation) and NMAA (number of accesses on attributes) showed good behaviour.

[8]This randomness should also express the interactive approach of such systems, which is guided by intuition rather than a systematic methodology, although experience has shown that at the beginning of a reverse engineering experiment we tend to apply a certain fixed set of graphs. This reflects the fact that the graphs address each a different level of abstraction.

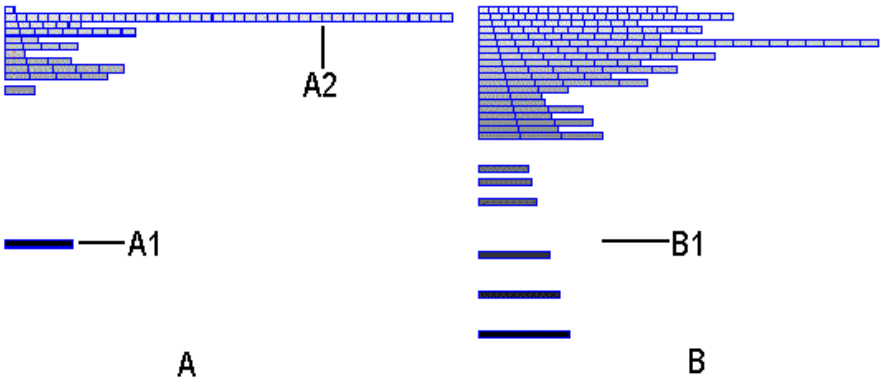


Figure 2.36: The coding impact graph applied on two classes of the Refactoring Browser. The width metric, as well as the color and vertical position metric is LOC.

Evaluation: This graph is very useful to *get a feeling* for certain classes or subsystems. It can show us what kind of implementation lies behind the subject entities and in certain cases what we should continue to explore.

2.2.11.3 Method Size Nesting Level

Graph	Checker, quadratic, sort according to width metric.	
Scope	Subsystem, especially inheritance hierarchy. No stand-alone classes.	
Metrics		
Size	LOC (lines of code)	NOS (number of statements)
Color	MHNL (hierarchy nesting level)	
Position	-	-

General Idea: A general rule is that big methods should be split up [BECK 97] into smaller chunks to increase their reusability and to make them easier to understand. This is especially true for methods that are implemented in classes deep down the inheritance hierarchy: perhaps parts of those big methods could be extracted and put up into a higher class to reuse them across several subclasses. The method size nesting level graph can help us to detect large methods deep down the inheritance hierarchy: It's a checker graph of methods with LOC and NOS as size metrics and MHNL as color metric. The nodes are sorted according to LOC, which puts the larger methods on the bottom area of the graph.

Since the color reflects the MHNL of the methods, we should be looking for big, dark nodes in the bottom area of the graph: these are possible split candidates. We call such methods split-and-push-up candidates.

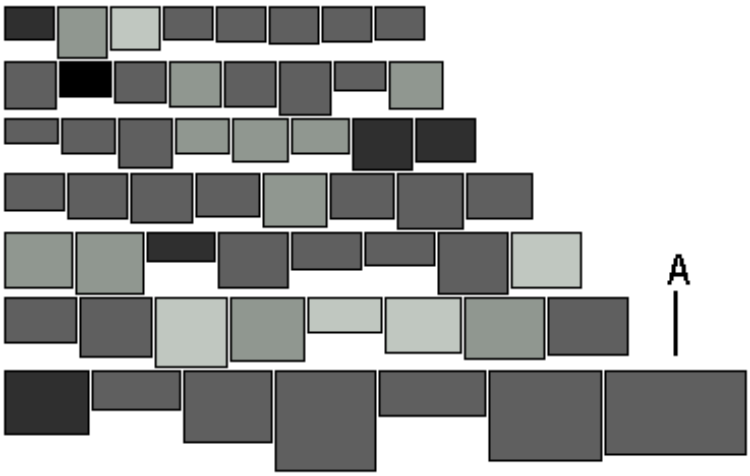


Figure 2.37: The method size nesting level graph applied on the largest Refactoring Browser methods. Size metrics: LOC, NOS. Color metric: MHNL.

Results with the Refactoring Browser: The Refactoring Browser shows in Figure 2.37 that is has been refactored itself a few times: there remain very few large methods, after filtering out all those with a LOC measurement smaller than 20. Yet, there are some large methods which also have medium MHNL values like those in the last row (A). Their lengths vary from 65 to 37 lines, which makes them also possible split-and-push-up candidates.

Results with Duploc: We display in Figure 2.38 only the methods that have more than 20 LOC and belong to non-stand-alone classes. The resulting graph shows us there are several very large methods, which on one hand don't have big MHNL values, but since they're not methods belonging to root classes either, are all the same split-and-push-up candidates. The biggest methods (A) have 201, 135 and 109

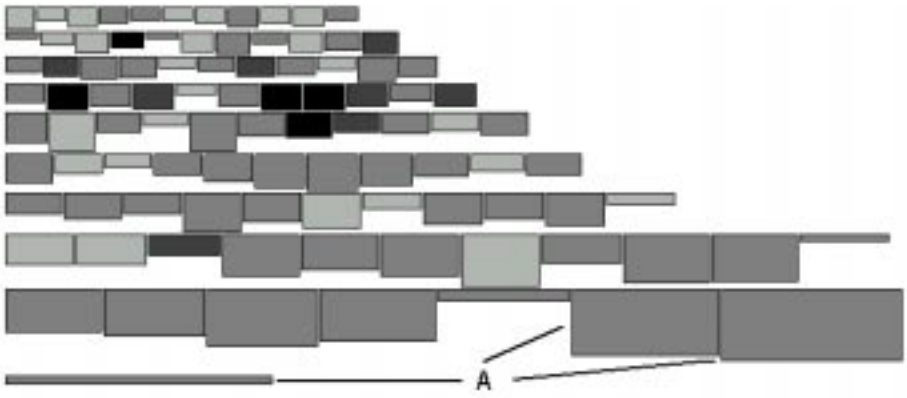


Figure 2.38: The method size nesting level graph applied on several Duploc methods. Size metrics: LOC, NOS. Color metric: MHNL.

LOC, which is way too much for Smalltalk methods. This excessive size is again due to the fact that most of them have never been refactored and written in one pull.

Possible Alternatives: The same graph using only LOC as size and color metric can be applied on whole systems (including stand-alone classes). In such a case the graph serves to easily detect very large methods which could be split up.

Evaluation: Since this graph is useful for classes belonging to inheritance hierarchies, it should primarily be used to get insights into such structures as to where the methods are which could be reengineering candidates.

2.2.12 Useful Graphs: Attribute Graphs

Attributes define the properties of classes. As such, it's mandatory that to understand the purpose of an attribute, we have to understand the class in which it is defined. This implies that very soon after applying one of the following graphs, we have to look at the source code of the class.

In this section we list the following graphs:

- DIRECT ATTRIBUTE ACCESS, Section 2.2.12.1.
- ATTRIBUTE PRIVACY, Section 2.2.12.2.

2.2.12.1 Direct Attribute Access

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem.	
Metrics		
Size	NAA (number of times accessed directly)	NAA
Color	NAA	
Position	-	-

General Idea: This is a graph of all attributes of a system or subsystem. As metrics we use NAA (number of times accessed) for the size and the color. We then also sort the nodes according to NAA. What we get is a clear display of which attributes are accessed the most in a system. These attribute nodes are positioned at the bottom of this graph. The largest nodes should be a case for closer inspection. The general rule should be that attributes which are accessed directly can break the system if the inner implementation of the attribute changes. This can be avoided by using an accessor method which returns the value(s) of the attribute. An accessor on such an attribute can provide a defensive wall of protection against such changes. There may also be some attributes which are never accessed and which may have been forgotten in the system and thus only add unnecessary complexity to it. They could be removed from the system. Such attribute nodes are positioned on top of the graph.

Results with the Refactoring Browser: In Figure 2.39 we notice at once that there is the attribute *class* (A) defined in the class *MethodRefactoring* which is directly accessed 86 times. We also see there are some never accessed attributes which should also be further investigated (B).

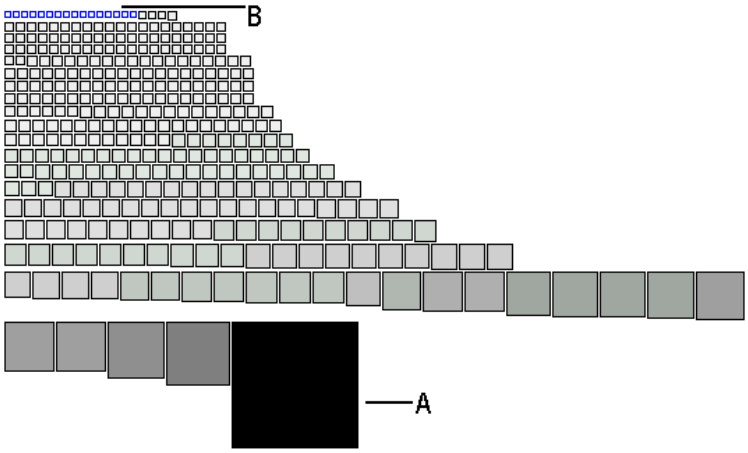


Figure 2.39: The direct attribute access graph applied on the Refactoring Browser. The size, color metric and sort criterion is NAA.

Results with Duploc: In Figure 2.40 we see that while in Duploc there are no attributes which are heavily accessed (the maximum is 31 direct accesses for the attribute *region* (A) defined in the class *AbstractRawSubMatrix*) there are many attributes which are never accessed (B) and which should be looked into for possible removal.

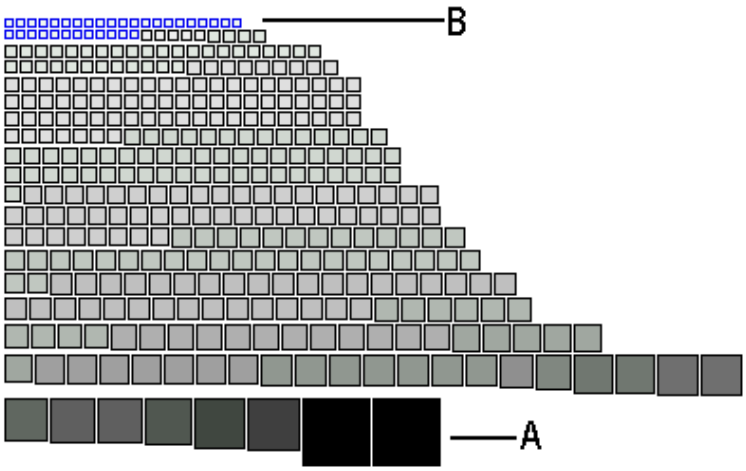


Figure 2.40: The direct attribute access graph applied on Duploc. The size, color metric and sort criterion is NAA.

Possible Alternatives: An interaction with interesting nodes is necessary to see if accessors have been implemented for them and if those accessor methods are used all the time.

Evaluation: A graph which works at every level of granularity. The next step which has to follow such a graph is to examine the classes in which the outlier attributes are defined. Note that this graph takes only the direct accesses into account. If an attribute is accessed very often through the use of an accessor method this will not show in this graph. Note that the quality of this graph depends heavily on the quality of the metamodel. Especially when building a model out of a CDIF file we have often seen that sometimes accesses are left out. This can lead us to wrong conclusion on never accessed attributes. Again, a check against the code has to be done to be sure.

2.2.12.2 Attribute Privacy

Graph	Checker, quadratic, sort according to width metric.	
Scope	Full system or subsystem. Better performance with C++ or Java.	
Metrics		
Size	NAA (number of times accessed directly)	NCM (number of classes which access this attribute)
Color	*	
Position	-	-

General Idea: Attributes may be directly accessed several times in a system. As we said in Section 2.2.12.1 such a situation is not ideal and can be detected with the graph described there. Apart from the number of times an attribute is accessed, another metric may prove to be useful for a similar graph: NCM, the number of classes which have methods that directly access a certain attribute. The attribute privacy graph is a checker graph which uses as size metrics NAA and NCM.

We are looking for wide, high nodes: such nodes are directly accessed a lot of times by many classes and should have an accessor at all costs, because the system easily breaks if such an attribute is tampered with.

Very wide but shallow nodes should also be looked at: although they are directly accessed a lot, it's by few or often only one class. If it's the case of only one accessing class, it should be checked it the attribute in question is private. If not, it can be made private without impact on the rest of the system.

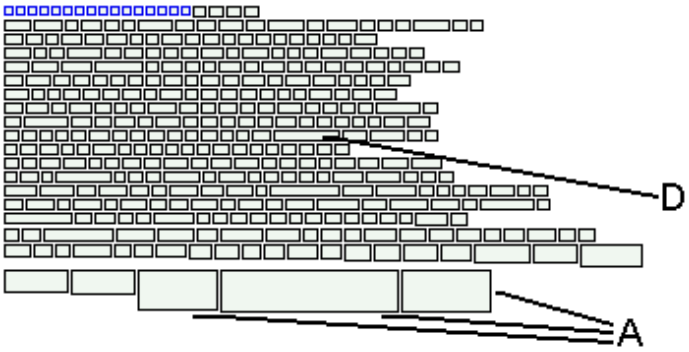


Figure 2.41: The direct attribute access graph applied on the Refactoring Browser. The size metrics are NAA and NCM.

Results with the Refactoring Browser: In Figure 2.41 we can spot some heavily accessed attributes marked as (A) which are accessed by many classes. We also see there are some very flat but wide nodes which are attributes heavily accessed by only 1 or very few classes.

Results with Duploc: In Figure 2.42 we can see that as a difference to the Refactoring Browser, Duploc has attributes which are seldom accessed by more than one class. The maximum NCM value is 3. We deduce from that that the implementor of Duploc keeps an eye on encapsulation[9].

Possible Alternatives: None.

[9] The implementor of Duploc used to implement a lot in C++, which could be a reason for the tight encapsulation.

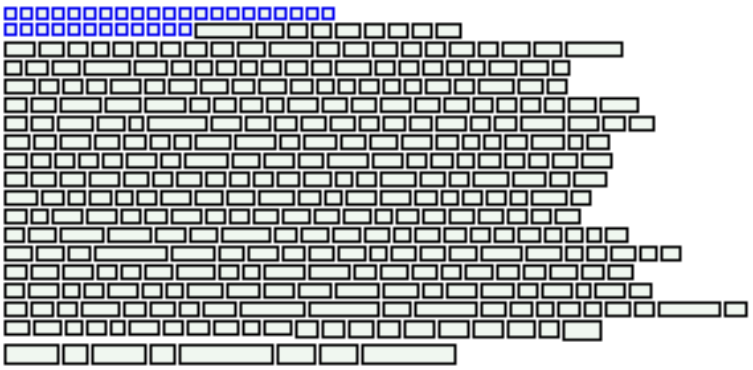


Figure 2.42: The direct attribute access graph applied on Duploc. The size metrics are NAA and NCM.

Evaluation: A graph whose purpose is to find attributes which have to be examined. Since such an examination takes place at textual level, it's a graph which can help find problems at once. The results are incomplete in this case: the last step after detecting wide and flat nodes would be to check if the attributes concerned are defined as private. If not they could be made private. However, this does not work in Smalltalk, so we had to leave that part out with our case studies.

2.2.13 Useful Graphs: Class Internal Graphs

A class internal graph treats the special case where the components of a class are displayed at the same time: methods and attributes.

In this case we find ourselves at a low level of abstraction, the source code is only one step away and it's necessary to look at it after applying a class internal graph.

In this section we list the following graph:

- CLASS COHESION, Section 2.2.13.1.

2.2.13.1 Class Cohesion

Graph	Confrontation graph, nodes sorted according to their width metrics..	
Scope	Single class.	
Metrics (Method Nodes)		
Size	LOC (lines of code)	NOS (number of statements)
Color	LOC	
Position	-	-
Metrics (Attribute Nodes)		
Size	NAA (number of times accessed directly)	NAA
Color	NAA	
Position	-	-

General Idea: This graph is a confrontation graph where the edges represent instance variable accesses between methods and attributes. This graph can indicate us how strong the internal cohesion of a class is. If a class has many accesses and looks very chaotic, this means that the class is difficult to split. On the other hand, if we can make out two or more separate clusters in this display, this is an indication that the class is a good split candidate. If the root class of an inheritance hierarchy shows such characteristics it is a sign that the hierarchy tends to be top-heavy. If the class shows sparse attribute accesses it could be easier to subclass.

Results with the Refactoring Browser: In Figure 2.43 we displayed the methods and attributes of the class *BRScanner* which has been identified as (C) in Figure 2.25. We gather at once that this class is heavily coupled internally and that splitting such a class is next to impossible.

Figure 2.43: A class cohesion graph applied on the class BRScanner. The method nodes (in the lower row) use as size metric NOS and as color metric LOC. The attribute nodes (in the upper row) use as color and size metric NAA.

Results with Duploc: We obtained some impressive results when we applied this graph to some classes of Duploc. We show only one here: the class *DuplocApplication*. After filtering out all methods that never accessed attributes, we got the graph displayed in Figure 2.44[10]. We clearly see two distinct clusters of attribute and method nodes. This class is thus certainly a split candidate. This suspect was confirmed afterwards when I asked the implementor of Duploc about this class. He confirmed that this class was to be split up during the next redesign of the system.

[10]Note that the graph resulted like this after direct manipulation of the graph (i.e. moving around nodes) and not because of a layout algorithm that can identify clusters. However, we included into CODECRAWLER the functionality to help us quickly identify such clusters.

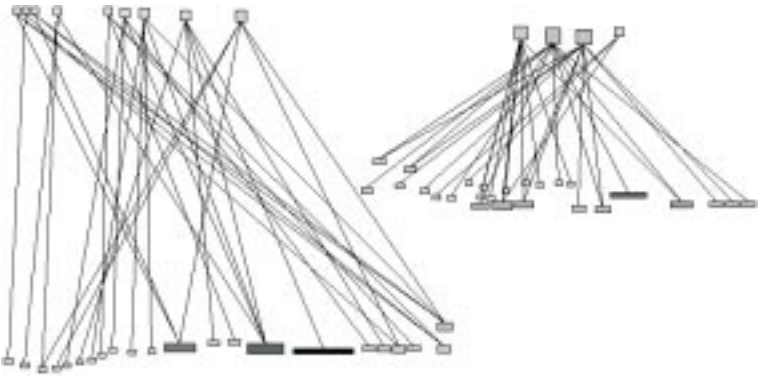


Figure 2.44: A class cohesion graph applied on the class DuplocApplication.

Possible Alternatives: We advise the user to remove all stand-alone nodes from the graph, as they are of no use in this case. The metrics, especially the color metric in the method nodes can be varied freely.

Evaluation: This graphs needs some interaction before it can express its full potential. However, its usefulness is indisputable: Up to this moment we haven't seen a technique which can detect split candidates with such an easy and quick method.

2.3 Grouping

Author: Oliver Ciupke

2.3.1 The problem

Reengineering large software systems requires analysing and manipulating tasks at multiple levels of abstraction. The standard views of a system used during design and implementation, such as the class and method structure as well as the source code, are often not sufficiently abstract. These views provide a wealth of information, not all of which is relevant to the task of reengineering. Consequently, the essential information may be obscured.

Additionally, some problems arising during the design of software are themselves only understandable at a high level of abstraction. For example, the class structure may look well designed even though there are too many dependencies between the packages or subsystems. This can make the system unnecessarily complex thus compromising flexibility and increasing the cost to maintain the system.

Abstract views of a system are already supported in several design languages and methods. Examples include packages in UML [RUMB 99] or subjects in OOA [COAD 91]. Unfortunately, they are currently to vaguely defined making them insufficient for the tasks occurring in reverse- and reengineering. Here, a concept must be implementable by tools in a way that it is composable with other techniques (such as querying and problem-detection) in a clearly defined manner.

2.3.2 Grouping

To be able to analyse a system, we must have an exact way to come from elementary views to more abstract ones. They must be formally sound, so it is known with precision what the meaning of an abstraction and to ensure that it is possible to build tools able to deal with these abstractions.

What we need is a description of how to move from a detailed description of a system to a more abstract view. We call this process *grouping* (also referred to as *abstraction*, *compression* or *lifting*). Grouping means replacing a set of entities, often describing a common abstract concept, into one abstract entity called a *group* (or complex entity).

Figure 2.45 shows a model of a small C++ program including classes and methods together with their relationships. Figure 2.46 shows a more abstract view on the same program where methods have been grouped together with the classes they belong to. One can imagine, that these kinds of abstractions are often useful to get an appropriate overview and to understand overall dependencies. For larger systems, it is even necessary to go beyond this level, e.g., to view it on the level of subsystems or packages.

2.3.2.1 Examples

In principle, any set of entities under consideration can be grouped together. Similarly, each entity can in principle be split up into its component parts in order to provide a more detailed view of the system. However, there are several groupings which are particularly useful in building up common abstractions. Examples for those are

Classes to packages: Grouping classes to packages (or subsystems or modules or clusters) is probably the application most commonly used. Figure 2.47 shows an example. The three upper left packages could be a framework, whereas the lower right package could be an instantiation of this framework towards a specific application. After grouping, dependencies between classes are propagated to the surrounding packages. The process leads to an overview over the overall architecture of the system.

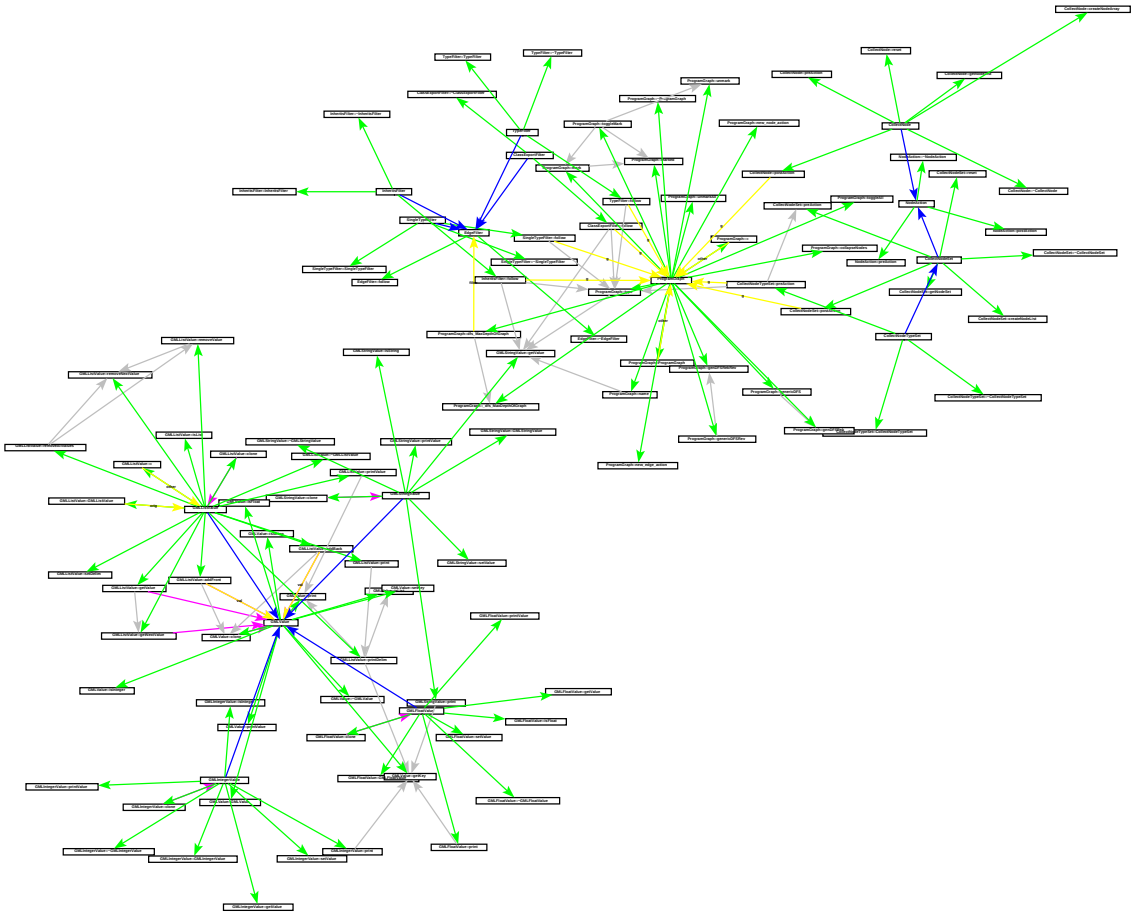


Figure 2.45: Program structure with classes and methods

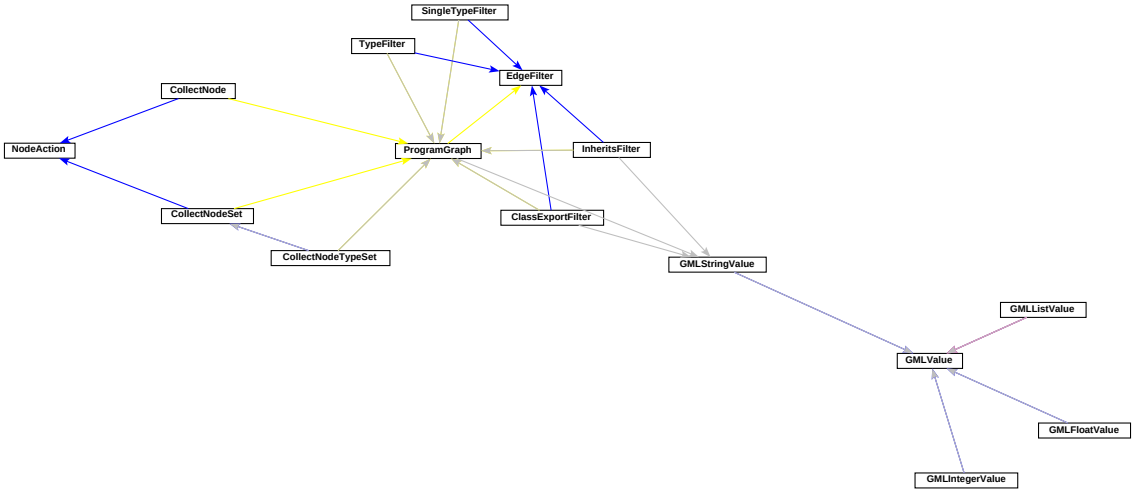


Figure 2.46: Program structure with collapsed methods

Interesting views in this example are as well those where not the classes of all packages (groups) are being replaced by a single node. We replace only those belonging to the framework itself, if we want to investigate how the application uses the framework. Or we could only group the instantiation if

we want to see, which classes of the framework are actually needed.

Packages to packages: In general packages may be grouped recursively giving a *grouping hierarchy*. In some programming environments, packages of different levels are given different names such as "subsystems", "program blocks", "service blocks" etc. Typically, this case occurs together with the former one (classes to packages).

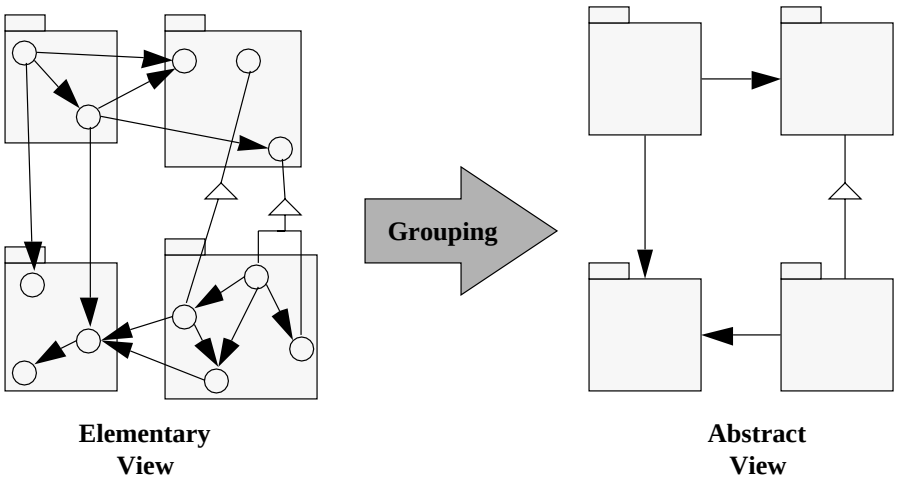


Figure 2.47: Grouping an elementary view to a more abstract one

Classes to super-classes: Grouping classes with (possibly one of their) their super classes is an abstraction commonly applied when modelling a system or an algorithm. An algorithm working on viewable elements is described in terms of an abstract super-class only.

Since we must take into account all possible dependencies, it is not enough only to consider only the super class, again we have to propagate relationships of all original classes to the remaining representant.

Objects to classes (or types): In a dynamic view, objects of a snapshot or a trace can be mapped to their classes. This is extremely useful due to the huge number of objects often being created during a program run. Therefore, similar functionality has so far often been implemented in tools such as visual debuggers (e.g., LOOK! [11] or profilers. Related work on visualising the behaviour of object-oriented systems are for example [PAUW 93], [KIME 94].

Dynamic method calls to pair of calling and called object: A set of messages between objects, i.e., method invocations during run time, can be grouped together with respect to caller and callee. If messages are modelled as associations between objects and represented by edges in a graph, then this is an example for grouping edges instead of nodes. It is often useful to label the resulting edge with the number of messages it represents.

Dynamic method calls within a certain interval of time: Messages can also be grouped with respect to given intervals of time. See for example [KOSK 96].

Classes to files and files to directories: In many programming environments, grouping classes to files and files to directories conforms to grouping classes to packages.

Objects to processes or physical machines: Objects can be grouped to their containing processes (or threads) or the physical machines they reside on. This grouping is useful, e.g., to examine the run-time performance of an concurrent (for processes) or distributed (for machines) object-oriented system.

[11] http://www.objectivesoft.com/ look.html

and many more.

The examples given above are applied in different contexts, mostly depending on the kinds of entities involved. Groupings containing *static* entities (classes, methods) are often used in model capture as well as forward and reverse engineering. *Dynamic* entities (objects, threads) are used when debugging or during performance or other dynamic analysis purposes. Groupings from or to *physical* entities (files, resources, databases, machines, processors) are needed for many development issues such as configuration management, checking consistency of compile dependencies, compile time analysis or documentation.

2.3.3 Definitions

We now want to provide an formal definition for our concept of grouping. First, we give a list of requirements which a concept should fulfill in order to be applicable in our tasks.

In the following, we will only consider grouping entities (nodes). Grouping associations (the elements of a relation, i.e., edges) can in principle be defined in an analogous way.

- The result of every legal grouping on every legal structure should be well defined.

  A tool implementing grouping should be able to give a meaningful result for any legal input.

- Entities should not just disappear. Every entity of a detailed view should be represented in an abstract view, though several entities of a detailed view may be represented by the same entity in the abstract view.

  A change in a method means in turn changing its containing class and package. For example, if an abstract view only represents packages, it must be possible to determine from the model which entities have been changed. Conversely, if we determine from the model that a package has to be examined then we must be able to find out which classes and methods are meant in detail.

- Dependencies should be maintained. If two entities in a detailed view obey a certain dependency, their representants should have a dependency of the same type.

  For example, if we change a methods interface in one package then we want to see from an abstract view, showing only packages, whether this change may affect other packages.

- Our concept of grouping should be well defined also in combination with other concepts such as filtering information.

Put together, these requirements already set up a kind of an informal definition or concept of grouping.

We represent the structure of a system as a *typed graph*. The entities of a structure and their associations are represented by nodes and edges respectively. A set of associations between a set of entities defines a relation on these entities. Nodes and edges may have different *types*. *Node types* are for example “Class”, “Method”, “Package”, “Object”; *edge types* for example “hasMethod”, “contains”, “creates”.

A group in terms of a graph which represents a system structure is just a set of nodes. Grouping is then replacing one or more groups (i.e., the corresponding nodes) by a node representing these groups while maintaining the edges between the nodes (see Figure 2.48). Formally, grouping and related terms are defined as follows. We use graph theoretic and relational algebraic terms in their usual definitions. Please note: for simplicity, we use the terms “Graph” or “directed Graph” here for what is exactly spoken a *directed 1-Graph* (see for example [HARA 69] or [SCHM 89]). This follows the terminology adopted throughout most of the related work.

- A *group* is a subset of all entities under consideration (including groups themselves).

- A *grouping* is a *surjective graph homomorphism* Φ mapping a Graph $G = (V, E)$ to a graph $G' = (V', E')$. The elements of a single group are being mapped to a single node, which represents this group.

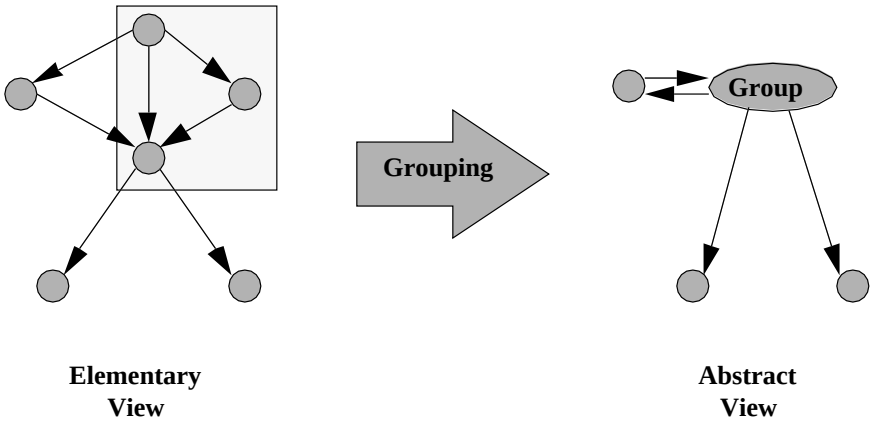


Figure 2.48: Grouping an elementary view to a more abstract one

- A certain grouping Φ can be given

  1. by a function H mapping the nodes of G to the nodes of G'. Since Φ is a graph homomorphism, it is completely determined by H. Due to the property of Φ as a surjective homomorphism, H is a *total, unambiguous and surjective relation*, i.e., a surjective function from V to V'.
  2. by its *spanning relation* S mapping every entity to be grouped to its surrounding group. S may be any *unambiguous* relation, i.e., partial function. We can compute H from S by

     $$H = (I \sqcap \overline{SL}) \sqcup S =: I \oplus S$$

     This is probably the definition most frequently used as input for tools, since it does not need to store nodes mapped to themselves. The notion of the overwrite operator for relations "$\oplus$" is borrowed from the specification language Z as described in [SPIV 92].
  3. by an *equivalence relation* A defining a group in terms of entities that are all equivalent.

     $$E = \{(x, y) | H(x) = H(y)\} = HH^T$$

     In this case, the grouping H is defined as the *canonical mapping* of A producing the *equivalence classes*

     $$V/A = VH$$

- The *trivial grouping* is the identity, which maps every graph to itself.
- The *cardinality* of a group is the number of its elements (since a group is a set).

For the ease of writing, we often identify these representations H, S and A with the grouping Φ itself in the remainder of this report.

2.3.3.1 Filters and views

Filtering is a concept complementary to grouping. Both grouping and filtering are needed to define a concept of views. While a surjective homomorphism defines a grouping, an injective homomorphism defines a filter. A view on a system can be defined in a formal way by a combination of groupings and filters.

- A *filter* is an injective graph homomorphism. A filtered view is a *partial sub-graph* of an other view.

A filter may select only a subset of the nodes and edges or restrict the graph to those nodes and edges fulfilling certain properties[12].

- A *view* on a model is the result of any combination (a sequence of concatenations) of groupings and filters applied to a model of a system. If a model is represented as a graph, then every view of this model is also a graph.
- The model itself is named the *complete view*.
- A view v_2 is said to be *more abstract* than another one v_1, if a non trivial grouping H exists, which maps v_1 to v_2, i.e., $v_2 = v_1 H$.

  This means that the process of abstraction is defined in a way that nothing just disappears, but details are (temporarily) hidden.

New views are produced from existing ones by sequences of abstractions and selections. In this way, abstraction and selection form operators which transform views. Since abstraction and selection can be used in different order, we will spend a short look on the properties regarding this matter.

Figure 2.49 shows an example where an abstraction and a selection are applied to a graph in different order. Unfortunately, the resulting graph differs depending on if abstraction or selection was first. In other words, abstraction and selection do not *commute* in every case.

The fact that the order of abstraction and selection matters is important for the interpretation of results of an analysis. Since abstraction only collapses information, while selection really omits it, abstraction should be performed first, if there is a choice.

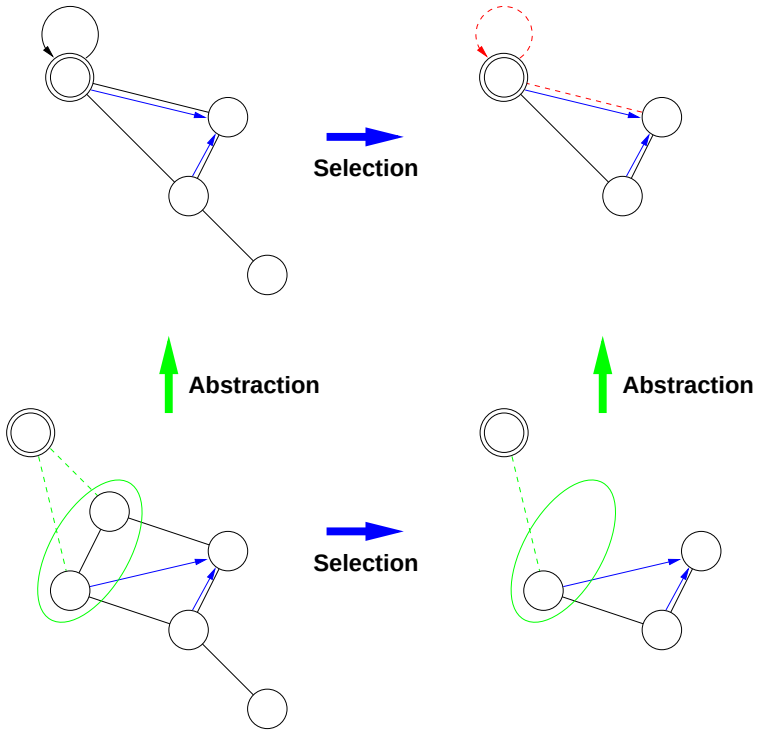


Figure 2.49: Abstraction and selection do sometimes not commute

[12] *Selection* and *restriction* are special kinds of filters on heterogeneous relations. The distinction is taken from relational algebra as used in database theory. To further confuse the situation in SQL a restriction is written within the “select” statement.

2.3.3.2 Group types

Every entity has a type (e.g., class, object, method, file). These are types on the meta-level and should not be confused with the types declared in the program or in the specification of the programming language.

The type of a group is determined by the possible types of entities it can hold (which may be more than one). Additionally, a group may be represented by an entity of a further type. For example the group of objects of a given class may be represented by this class (though it is not equal to this class).

2.3.3.3 Group operations

A model of a system may provide operations on its entities or relations. For example a user may want to click onto a node to see or edit its properties, e.g., the source code it represents. For groups, there are additionally special operations available:

collapse: replace the set of entities contained in the same group by a representation of this group (e.g., in a certain view)

expand: replace the group by its elements (e.g., in a certain view)

To perform an operation on each single element of a group or to filter a subset of elements from a group fulfilling certain properties (i.e., a predicate) there are well known operations from functional programming [BIRD 88]:

map: takes an operation o and a set (or group) s and performs o on each element of s[13]

filter: takes a predicate p (a function returning true or false) and a set (or group) s and returns a set s' containing those elements x of s, for which $p(x)$ is true

2.3.3.4 Visualising groups

A major goal for the concept of grouping is to support high level visualisations. There are different possibilities, how to realise such visualisations.

Since grouping produces views which are simply graphs, the visualisation of the result of a grouping is straightforward. Groups that are collapsed are shown as a node replacing the elements contained in the group. Relations of those elements are propagated to the group as required by grouping as a graph homomorphism.

But we also want to show the original situation. There we want to see all detailed information, but we also want to see which are the groups and which entities belonging to the different groups. Groups can also be visualised when they are not collapsed to a single node, but when they are still (or again) expanded and all their elements are present. This is especially important when groups are defined interactively by the user. Possibilities for such visualisations are

- Drawing a shape (e.g., a box) around the elements to be grouped. This requires all elements of a group to be localised near each other. This is for example the solution defined in the UML for packages. We drew groups in this way in Figure 2.48.
- Drawing all the entities contained in a group in the same way, e.g., in the same colour or with the same shape.
- The group is shown as an additional node which is connected to its elements by its defining containment relation. This way, elements of the detailed view and the abstract view are shown at the same time. We used this visualisation in Figure 2.49.

[13] Sometimes "*map*" is referred to as "*foreach*", but this term is often used with a different meaning in the area of concurrent programming.

2.3.4 Tool support

Our tool set GOOSE already supports several common use-cases requiring grouping functionality. Most of it is implemented in the program REVIEW for querying and manipulating graph data in GML format.

REVIEW reads a graph from input, a set of commands is being performed on the graph, and result is given as output. Several commands are available to deal with grouping:

collapseList Collapse a list of nodes to a new node. The command reads a file that contains the name of every node that is to be collapsed. Each line has to contain one node name. The first line is the name of the new node to create. The collapsed nodes are deleted from the resulting graph.

The new node replaces the deleted nodes in all edges, i.e., the new node is connected with another node of the graph, iff one of the nodes in the given list was and that node is not in the given list.

collapseMethods Every node in the graph of type *Method* are collapsed with the class node that they belong to. The method nodes are deleted and the class node replaces the deleted nodes in all edges. So this command abstracts from methods but keeps the relations between classes through methods.

collapseSubsystem All nodes that belong to a subsystem are collapsed with the subsystem node. (Similar to collapse nodes.)

collapseSubsystemList Reads a list of subsystem node names from file. Each line has to contain one name. REVIEW invokes *collapseSubsystem* on each name in the file.

addHierarchy Adds a hierarchy to the program graph. The hierarchy is created by subsystem nodes and *consistsOf* edges. It forms a tree where the leafs are class nodes.

makeSimple If there is more than one edge between any node, then replace them by only one edge, even if the two edges are of different types. This command is useful for graph layout algorithms that require simple graphs, i.e., graphs with at most one edge between nodes.

makeSimpleTypes If there is more than one edge of the same type between any node, then replace them by only one edge of the same type. Use this command, if you don't care about the individual relationship but about the fact that there is a relationship of a certain type.

All but the last two commands group nodes. The last two commands implement two frequently needed edge groupings.

To give an example, Figure 2.46 was derived from Figure 2.45 by REVIEW by using the commands:[14]

```
collapseMethods
deleteSingles
makeSimpleTypes
```

Another useful tool is a script analysing a directory structure and producing a subsystem hierarchy as required by the `addHierarchy` command. This is applicable in programming environments where the subsystem structure is denoted by the structure of files and directories [RITZ 98]. This is for example the concept used by Java in most cases.

A tool providing visualisation for groupings which conform to the UML notation has also been implemented within the context of this project, see [KOSK 98a].

[14] In fact during normal usage, all these commands are invoked automatically by a script.

2.3.5 Open issues

Some issues are still unresolved and need further research.

Grouping edges It is also possible to group edges. Grouping edges occurs far less frequently but may be useful in several situations. All edges to be grouped are mapped to the edge representing the group. In most cases edges between the same nodes (called parallel edges) are being grouped, e.g., function calls between the same classes. There is no clear definition of how to group non adjacent edges, yet. Adjacent nodes may be grouped too (may be ambiguous in non-directed graphs) or such edges may be mapped to hyper-edges. We will delay a clear definition until need arises from applications or tool development.

Our tool REVIEW currently supports grouping together all edges between the same nodes having the same type.

Non-hierarchic grouping A non-hierarchic grouping is a grouping defined by a non-hierarchical relation. A non-hierarchic grouping contains groups which share at least one element. E.g., two packages may share the same class. Non-hierarchic grouping introduces a "shared containment" relation. Groups are connected by this relation, if they both contain the same element. A non-hierarchic grouping is not any more defined by a homomorphism, since a non-hierarchical relation is not a function.

It is not fully clear yet, which are exactly the cases where non-hierarchical grouping is needed. E.g., a shared class could also be moved to a third package with relations to this package. One example is to split up a system with respect to functional responsibilities (communication, user interface), where entities may serve for several functionalities.

In practical applications, there are both examples explicitly forbidding or allowing non-hierarchic grouping:

- A package structure must be organised hierarchically in Java.
- In contrast, the object modelling method described by Coad and Yourdon [COAD 91] allows different packages (in this context named "Subjects") to share the same classes.

Another open question is whether shared elements must propagate all their relations to all groups containing them or if these relations may be split up between groups.

2.3.6 Summary and future work

We presented the concept of grouping which can be applied for both static and dynamic structures as they are often used in the context of reengineering software systems. It provides a formally sound way to deal with views on different levels of abstraction. We provided formal definitions for grouping, filtering and views. Filtering information is a concept complementary to grouping. Consistent views on a system result from combinations of grouping and filtering. We gave examples for different ways to visualise abstract views, and gave a brief overview of our tools supporting different tasks related to grouping.

Some of the tasks within this topic still require further work. Among them are:

- Grouping is already being supported by our tool prototypes, but there is still a wish-list of improvements. For example, more flexible and reusable methods for the specification of groupings and support for complex entities within the visualisation.
- A further look has to be taken on some formal issues which are still open, as mentioned in Section 2.3.5. Some clarification to these questions will also have to come from further practical usage in case-studies.

- A lot of knowledge exists about what “good” design means on the levels of abstraction as they are usually used in forward-engineering (e.g., the class level). For large systems it is important to gain similar kind of knowledge on higher levels of abstraction, such as the subsystem or package levels. It has to be investigated, how this can be supported by our comprehension of grouping.

2.4 Reorganisation

Author: Benedikt Schulz

In this section we present a new technique for the reorganisation of object- oriented systems. The key idea of our approach is introduce design patterns into a system through the application of refactorings. The use of design patterns leads to a more flexible design, whereas the application of refactorings ensures the correctness of the transformations.

The section is organised as follows. In section 2.4.1 we describe the fundamentals of our approach involving design patterns and refactorings. These two techniques are combined into a powerful technique for reorganisation in section 2.4.2. Our approach is discussed and evaluated in section 2.4.3. We present related work in section 2.4.4 and conclude in section 2.4.5 with suggestions for further work.

2.4.1 Fundamentals

Changing a system by hand is an extremely time-consuming, difficult and error-prone task. Consider the case of simply renaming a method of a class. After the name has been changed, all places where this method is called, which can be spread over the whole system, have to be identified and changed accordingly. Because of these difficulties, a *technique for reorganisation should allow for tool-support.*

The aim of reorganisation is not to add new functionality to a system, but to make it more flexible in order to make it easier to add the desired new functionality. Therefore the reorganization should not change the behaviour of the system. Thus the *technique must define the notion of system behaviour making it possible to prove, that a performed reorganisation-step does not change this behaviour*.

In this section, we present two approaches for the reorganisation of object-oriented systems. In 2.4.1.1 we describe the high-level approach first presented in [ZIMM 97]. This approach describes design patterns as operators rather than as building blocks. The low-level approach of William Opdyke, who describes in [JOHN 93] a set of transformations he calls *refactorings*, is presented in 2.4.1.2.

2.4.1.1 Design Patterns as Operators

Design patterns have recently gained much attention in the field of software engineering. Many people consider them to be a promising approach for overcoming some fundamental problems in the design and reuse of object-oriented software. Design patterns were first introduced to the software community by Gamma et. al [GAMM 95], [GAMM 91] in the early 90s. They are descriptions of solutions to a set of common, recurring design problems within a particular context. They consist of a pattern name, a problem description, the presentation of a well-proven solution and the consequences and trade-offs of applying the pattern. The essential advantages of the application of design patterns are:

- Better software design can be achieved via the reuse of well-proven solutions for general design problems.
- Design patterns form a standardised terminology for the modelling of software systems. They ease the documentation of design decisions and make them more understandable.
- Design patterns are language independent.

Over the years, some weaknesses in design patterns have been identified. One major problem is the absence of a more formal classification of design patterns and their mutual relationships [ZIMM 95]. Due to the commonly very informal description of the purpose and the structure of a design pattern it is often hard to choose the right pattern variant.

Another major problem is the lack of a systematic way to integrate design patterns into existing systems. The descriptions of design patterns are so far mainly focused on the depiction of the target structure resulting from the application of the design pattern. The process of pattern application itself is often not taken into consideration. That means that up to now, design patterns have been seen as building blocks [BUSC 96], rather than as operators. This allows for the easy construction of new systems, but makes it very hard to introduce design patterns into existing systems in order to reorganise them.

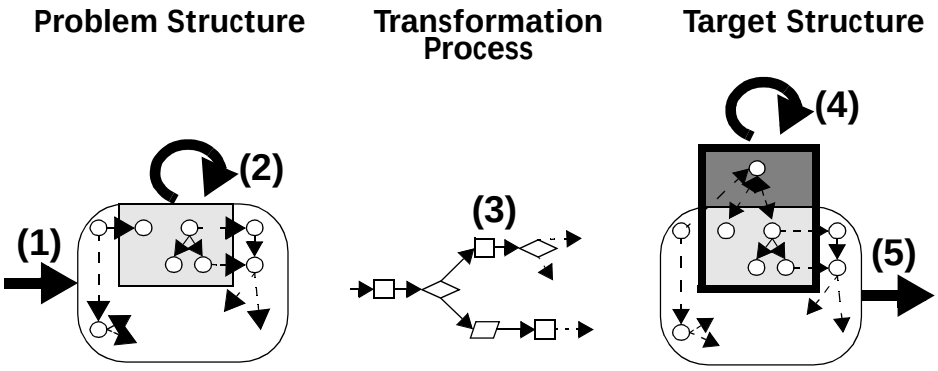


Figure 2.50: Systematic process of design pattern application

In order to overcome these shortcomings, a new approach is presented in [ZIMM 97]. The author introduces a novel concept to support the systematic application of design patterns to existing software systems. In his approach he considers design patterns to be operators, whose application transforms an existing design into a target design, fulfilling the necessary requirements. In order to make a systematic approach possible, the process of design pattern application is split into five steps (see Figure 2.50). The example given in Figure 2.51 and 2.52 illustrates the algorithm of applying the pattern *Bridge* to an existing design:

1. **Identification of the problem structure**: The software engineer identifies the part of the existing design that should be reorganised. In the given example (see figure 2.51), the application of operator *Bridge* aims at de-coupling the abstraction (superclass `Set`) from its implementations. The problem structure comprises the class `Set` and its descendants.

2. **Check of pre-conditions**: All pre-conditions must hold in order to apply the design pattern operator. In our example, the implementations `HashSet` and `ListSet` must provide the same interface as the abstraction `Set` in order to be interchangeable.

3. **Parameterised transformation of the problem structure into the target structure**: In this step the design pattern application itself is put into place. The process is divided into generic aspects (depending on the design pattern) and application specific aspects (parameters). The user parameterises the process by choosing an appropriate variant of the design pattern. Additional parameters are for example the set of methods which will be delegated or the decision whether the class of the implementation object shall be interchangeable at run-time or not.

   In our example, a new abstract class `SetImpl` (the new implementation superclass) is introduced. A new attribute `impl` is added to the class `Set` (**3a**). The methods of `Set` are divided into two groups; implementation-independent methods which remain in `Set` and implementation-dependent methods which are moved to `SetImpl`. This division is performed by the user. `Set` delegates all calls to implementation-dependent methods to `SetImpl` (**3b**). Now the user may decide whether he wants the implementation objects to be interchangeable at run-time or not (**3c**). In our example, the method `changeImpl` provides the corresponding functionality.

4. **Reorganisation of context**: The transformation of the problem structure often causes a change to its interface. The clients of the interface affected by the change have to be reorganised in order to provide the same functionality as before. In our example all calls of the form `new(HashSet)` are replaced by `new(Set(HashSet))`.

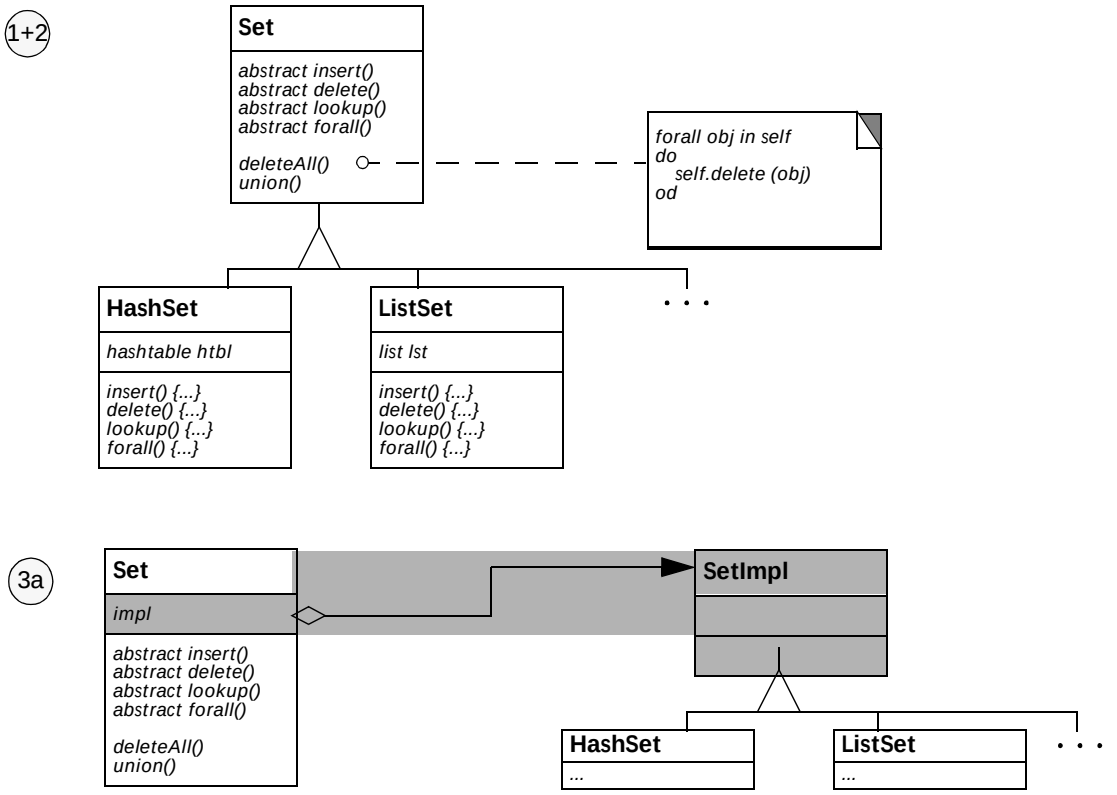


Figure 2.51: Application of Design Pattern Operator *Bridge* (I)

5. **Check of post-conditions**: Post-conditions must hold after the design pattern has been applied. It is often hard to formalise post-conditions as they usually depend on semantics. Therefore they are often described informally. In our example the condition *"the class of the implementation object is interchangeable at runtime"* must hold after the application of the operator *Bridge*.

The needed transformations are described in terms of meta model operations. For this purpose, a meta model comprising the entities *Class*, *Method*, *Param*, and *Attribute* and a description language are introduced. This language consists of three sets of constructs:

- transformation operations, e.g., `applyDelegation(<from>, <to>, <Params>)`
- parameterisations by the user, e.g., `<InputVar>?:<VarType> [<InitList>]`
- miscellaneous operations, e.g., conditions and labels

With the approach described above, the application of design patterns becomes an algorithmic process. In the "building block" approach, the user must know about and understand the informal description of the design pattern and all its relationships and variants in order to apply the pattern to the design. The explicit separation of generic and specific aspects (e.g., parameters) of the design pattern and the operational presentation simplify its application, since the user only has to cope with use-case specific aspects of the design pattern application. Beyond this, the transformational approach enables to use design patterns as *reorganisation operators* in order to reorganise existing systems by identifying the problem structure and transforming it into an improved target structure using the algorithm described above.

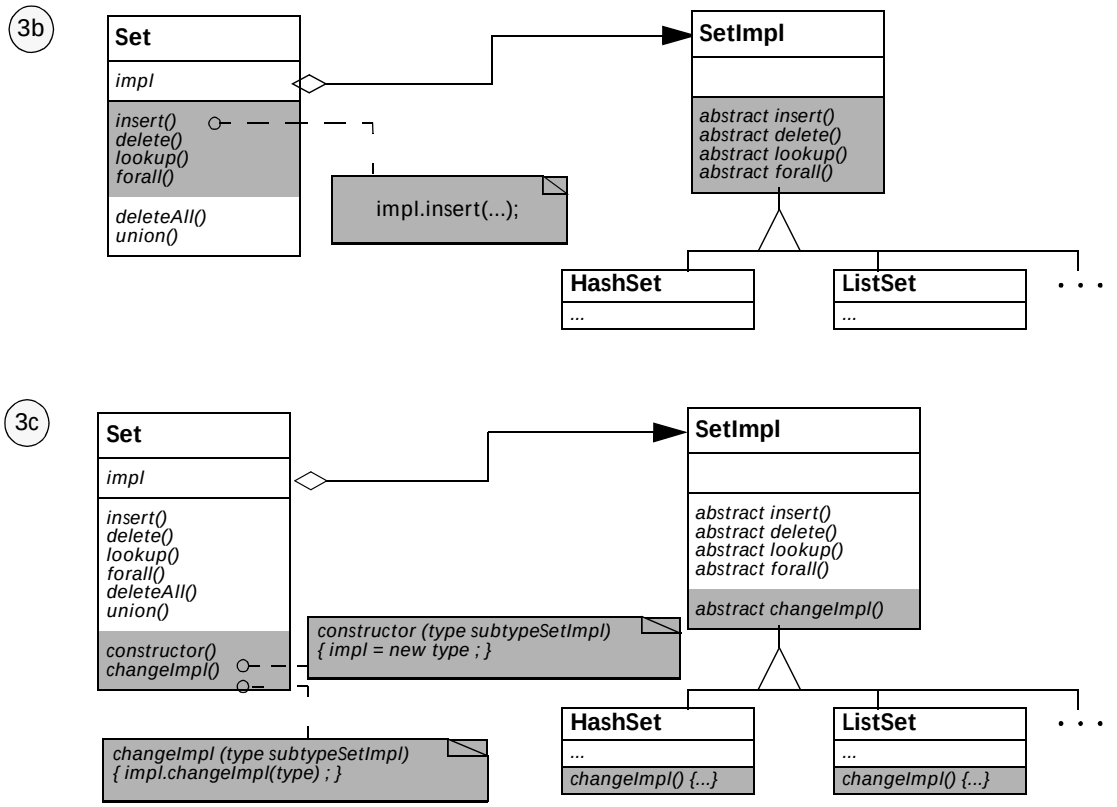


Figure 2.52: Application of Design Pattern Operator *Bridge* (II)

2.4.1.2 Refactorings

In his PhD thesis [JOHN 93], William Opdyke presented an approach for the support of the evolution of object-oriented systems. He defined a set of so-called *refactorings*, which are implementable in a tool and showed that the application of these refactorings does not change the system behaviour.

Opdyke calls the property of not changing the behaviour of a program *semantic equivalence* and defines it as follows:

> Let the external interface to the program be via the function *main*. If the function *main* is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same.

Besides the semantic equivalence, Opdyke additionally defines several more invariants to ensure the syntactic and semantic correctness of the refactorings. He does not, for example, allow the use of *multiple inheritance* or the *redefinition of inherited member variables.* Every refactoring has to preserve these invariants.

The refactorings are divided into two subsets: One subset contains 26 *low-level refactorings* focusing on simplicity in order to make it easy to prove that system behaviour is not changed. The other subset consists of three *high-level refactorings*, which are defined by using the low-level refactorings. The proof of the behaviour preserving nature of the high-level refactorings is also furnished in terms of low-level refactorings.

Low-level refactorings allow the creation (e.g., `create_empty_class`), deletion (e.g., `delete_member_function`) or modification of program entities (e.g. `add_function_argument`).

Furthermore, refactorings to move member variables (e.g., `move_member_variable_to_superclass`) and simple composite refactorings (e.g., `convert_code_segment_to_function`) belong to the low-level refactorings. The description of these refactorings is as follows:

- **Name**: Denotes the name of the refactoring.
- **Description**: Describes the refactoring and its variants.
- **Arguments**: Enumerates the set of arguments and their meaning.
- **Pre-conditions**: Lists the set of pre-conditions, which have to hold in order to assure that the semantics of the program remain unchanged when applying the refactoring.
- **Proof of correctness**: A semi-formal proof that ensures that the refactoring does not change either the invariants or the semantics of the program, assuming that all pre-conditions hold.

The three high-level refactorings described in [JOHN 93] are:

- **Creation of an abstract superclass**: The common abstraction (methods, member variables) of two or more classes is factored out into an abstract superclass.
- **Subclassing and simplifying conditionals**: Conditional statements, dependent on the internal state of the object, which are used to select the different behaviour of an object are replaced by an implementation using various subclasses and polymorphism.
- **Converting an inheritance relation into an aggregation**: A relation modelled by inheritance is converted into aggregation. The functionality, which was inherited can be accessed by delegating it to the component.

The description of the high-level refactorings contains one additional section describing how it is decomposed into low-level refactorings, how the pre-conditions can be derived from them and how the proof of correctness is built upon the proof for the low-level refactorings.

Since Opdyke presented the pre-conditions as well as the necessary restructuring in a detailed way, it was possible to construct tools for the support of the refactoring task. Due to the complexity of the C++ language [15] Ralph Johnson and his group focused on Smalltalk and constructed a *Refactoring Browser for Smalltalk* [ROBE 97b]. This tool supports a selection of the refactorings described in [JOHN 93] and was successfully used for the development of *Hotdraw*.

Berthold Mohr showed in [MOHR 98] that it is possible to implement a tool supporting refactorings for a subset of C++. He implemented the refactoring for the conversion of an inheritance relationship into an aggregation including all necessary low-level refactorings.

In spite of the promising results in formalising [JOHN 93] and implementing [ROBE 97b][MOHR 98] refactorings, there are some restrictions to the approach:

- **Behaviour preservation**: The described approach of assembling high-level refactorings from low-level refactorings requires behaviour preservation for every application of a low-level refactoring. It may be desirable to ignore this constraint and to guarantee behaviour preservation for the entire high-level refactoring only.
- **Language dependance**: It is not possible to specify all the pre-conditions in a language-independent way, because the semantics of a program heavily depend on the semantics of the underlying programming language.

[15] Opdyke writes in [JOHN 93]: "The C++ language is a semantically complicated language, supporting machine level operations such as pointer arithmetic; these complexities make it difficult to more precisely define what behaviour preservation means for C++ programs."

- **Pre-condition checking**: Checking whether a pre-condition holds or not, is a difficult task which requires global dataflow analysis techniques. This is, in general, an unsolvable problem.
- **Level of abstraction**: Even the high-level refactorings are not on a level of abstraction which is necessary for reengineering.

Especially the inappropriate level of abstraction is annoying when refactorings must be used to re-engineer large object-oriented systems. We present a new approach to overcome this shortcoming in the following section.

2.4.2 The FAMOOS approach

This section will begin with a definition of a set of requirements a reorganisation methodology must meet. Then we show that the approaches presented in section 2.4.1 are not adequate to fulfil these demands. In the last part, we introduce a new methodology for the reorganisation of large object-oriented systems.

We have identified the following requirements which must be fulfilled by a reorganisation methodology in order to be practically applicable for large object-oriented software systems:

- **Language independence**: The methodology should not rely on a specific object-oriented programming language.
- **Level of abstraction**: The methodology should aim at the design level rather than the implementation level, since most of the problems of an object-oriented system which can be solved by reorganisation (e.g., lack of flexibility) concern the design of that system.
- **Behaviour preservation**: The goal of reorganisation is not to change the functionality of a given system, but to improve the structure of that system in order to make these sorts of functionality changes easier. Therefore a reorganisation operation should not change the systems behaviour. The methodology must be proven to preserve behaviour.
- **Tool support**: The methodology requires for tool support, since reorganising a large system by hand is error-prone and time-consuming.

In the preceding section, we presented two different approaches to the reorganisation of object-oriented systems. Although both of them are very promising, they suffer from some weaknesses which make it difficult to put them into practice. While design pattern operators fulfil the first two requirements, (i.e., they are language independent and deal with design aspects) they fail to fulfil the last. Thus, not only is the underlying model too abstract to be implemented in a tool, but the pattern operators also lack proof of behaviour preservation. Unlike low-level refactorings, the atomic operations which make up a design pattern operator do not necessarily lead to behaviour preserving states. In the example given in Figures 2.51 and 2.52, the application of step 3 changes the behaviour of that system as constructs of the form `Set s = new(HashSet);` are no longer valid due to the broken inheritance relationship. On the other hand, refactorings provide at least a semi-formal proof of behaviour preservation. Even more, it was shown that they can be implemented in tools [ROBE 97b], [MOHR 98]. However, refactorings, even the high-level refactorings are not at a level of abstraction which is needed for reorganisation. Beyond this the specification of refactoring-specific pre-conditions heavily depends on the semantics of the underlying programming language. Thus it is not possible to use the refactoring-approach in a language independent way.

Our approach offers the best of both worlds. We combine the two approaches in order to overcome the shortcomings of both of them by *implementing the design pattern operators* [ZIMM 97] *with Opdyke's refactorings* [JOHN 93].

Whereas in [ZIMM 97] the description of the operators is very informal, we are now able to formalise the operators, as well as their pre-conditions and post-conditions with the aid of the refactorings. This

means, that we replace the meta-model and the operator-language defined in [ZIMM 97] with the model and language defined in [JOHN 93].

The different phases for applying a design-pattern operator in [ZIMM 97] are refined as follows:

1. **Identification of the problem structure**: As in [ZIMM 97] the software engineer has to identify the part of the system to be reorganised first. Depending on the selected problem structure and the design pattern operator to be applied, the sequence of refactorings can be determined.
2. **Checking the pre-conditions**: The preconditions are derived from the sequence of refactorings. They are checked before each application of a single refactoring. However it is possible to derive a set of preconditions which, when satisfied, ensures the success of the application of the whole sequence.
3. **Parameterised transformation of the problem structure into the target structure**: This step performs the transformation by executing the sequence of refactorings. As in [ZIMM 97] this may require a parameterisation by the user.
4. **Reorganisation of context**: Since the reorganisation of the context is part of the application of a single refactoring, the context of the problem structure does not need to be reorganised after the execution of the sequence of refactorings.
5. **Checking the post-conditions**: The post-conditions hold after the application of the design pattern, because every refactoring preserves the given invariants, while at the same time not changing the behaviour of the system. An explicit checking of the post-conditions can thus be omitted.

Figures 2.53 and 2.54 depict the third step of the application of the design pattern operator *Bridge* as performed by our new approach. The sequence of necessary transformation steps is different from the sequence in [ZIMM 97], since our approach requires behaviour preservation at every step.

In **step 3a** a new, unreferenced class `AbstractSet` is introduced. `AbstractSet` is assigned to the same superclass (not shown in the figure) as `Set` in **step 3b**. The introduction of a component relationship between `Set` and `AbtractSet`, the copying of the interface of `Set` to `AbstractSet` and the delegation of all methods of `AbstractSet` to its component `Set` is depicted in **step 3c**.

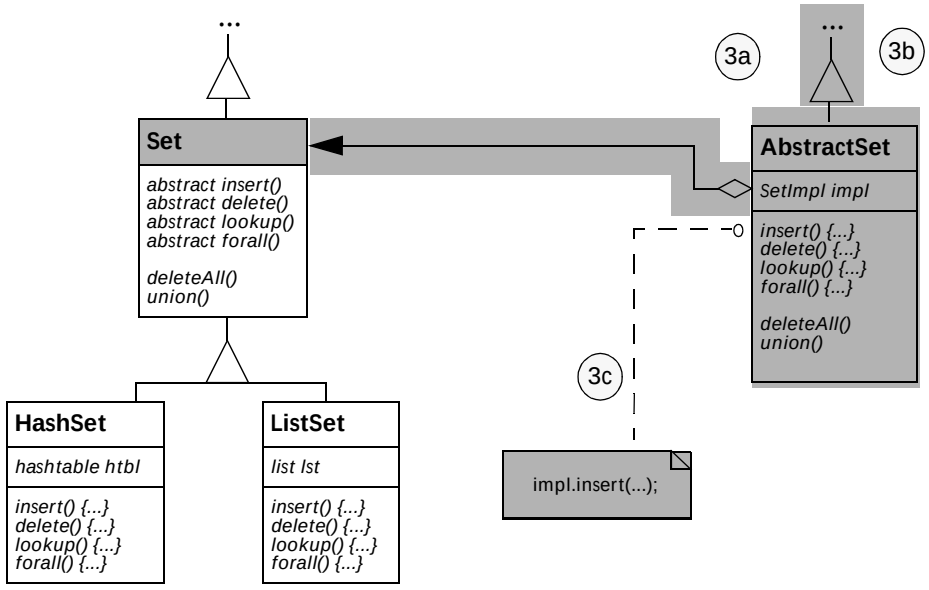


Figure 2.53: Application of a *Bridge* with refactorings (I)

Step 3d as depicted in Figure 2.54 contains the most crucial refactoring. First the constructor method of `AbstractSet` is changed to include one parameter determing the actual implementation (e.g., `HashSet` or `ListSet`) of the set. Then every construction of a set (subclass of `Set`) is changed to use the new constructor of `AbstractSet` and every variable definition using `Set` or one of its subclasses as a type is changed to `AbstractSet`. Finally in **step 3e** a new method `getImpl` can be introduced in the class `Set` to determine the actual implementation class of the set.

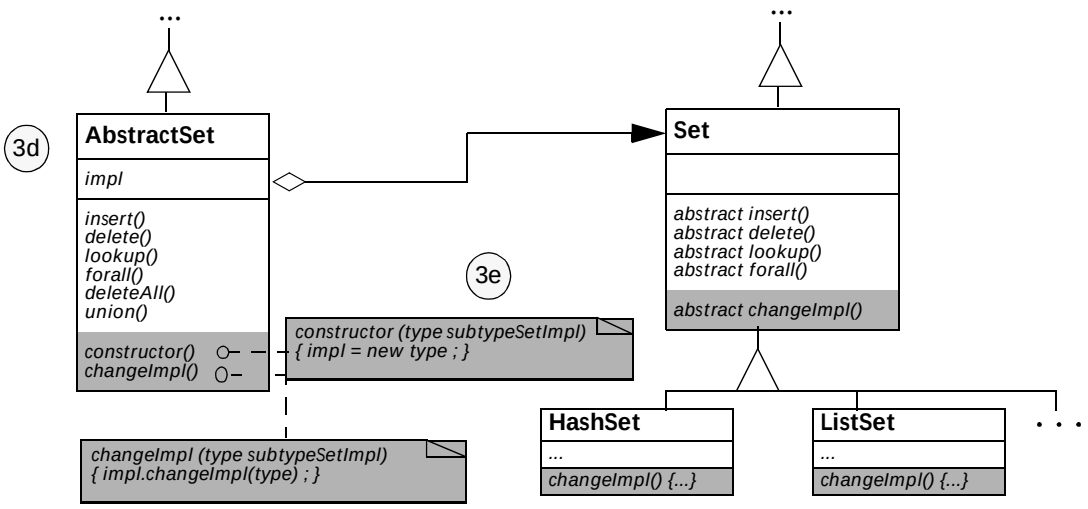


Figure 2.54: Application of a *Bridge* with refactorings (II)

Our approach fulfils all the already mentioned requirements. The method is **language-independent** on the design-level, because design patterns are language-independent. The design level is the **appropriate level of abstraction** for a reengineering task. The application of design pattern operators is correct in the sense of **behaviour preservation**. Since every single primitive transformation is fully specified together with pre-conditions and post-conditions, it is possible to implement **tools to support our approach**.

2.4.3 Discussion

In this section we present some results of the application of our approach to an existing system. For this purpose we implemented a tool prototype which provides several simple or complex refactorings for C++ (e.g., add_class, change_superclass, convert_inheritance_into_aggregation), [MOHR 98] using the FAST C++-parser [Sem 97]. The sequence and order of refactorings required to implement a particular design pattern is determined via shell scripts. We tested our tool with an existing software application for the visualisation of flow data from the field of hydraulic engineering. The aim of our experiment was to make some hot-spots in the system more flexible in order to ease the functional extension of the software. Some of our experiences are described below.

The software system mentioned was developed for a Windows NT platform using a proprietary graphic library providing several drawing classes (e.g., `Shape`, `Circle`) and implemented using the Windows Graphic Device Interface (GDI) as a basis. Due to performance aspects, with a new version of that software, it was decided to use the Windows DirectDraw interface. In order to be compatible with older versions of Windows NT, one requirement was that the platform-dependent graphic sub-system should be interchangeable.

In the original design the platform specific graphic output routines (e.g., `drawPoint`) were coded in `Graphic` using GDI-routines. All graphical shapes inherited the needed base functionality from `Graphic` (see Figure 2.55). Therefore it was hard to introduce new functionality to support more than one graphic sub-system, since the platform-dependent parts were strongly coupled to the domain-specific

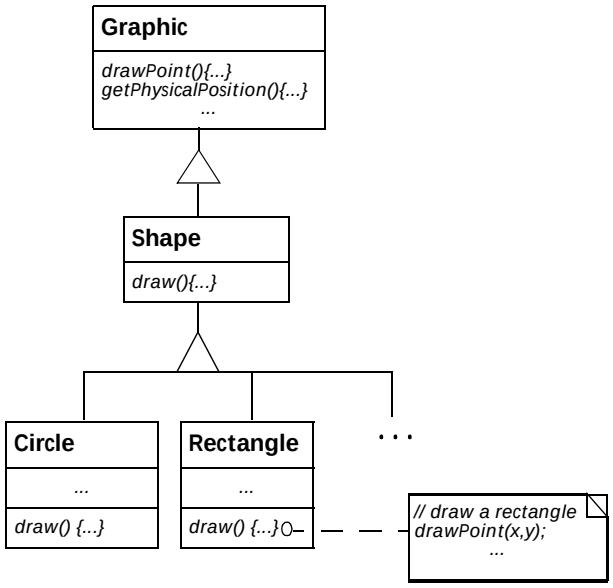


Figure 2.55: Problem structure before reorganisation

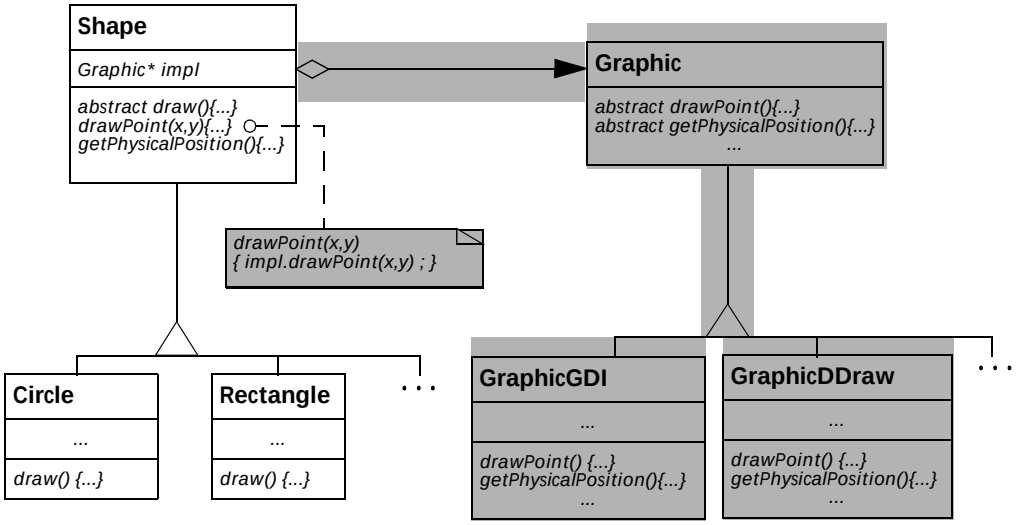


Figure 2.56: Improved target structure after the reorganisation and functional extension

classes. On the other hand, the relationship between `Shape` and `Graphics` was equivalent to an *implementation inheritance* and `Graphic` was never used as a static type of a variable or in a polymorphic way throughout the entire system.

One way to solve that problem is to *objectify* the parts of the problem structure which should be interchangeable. We decided to apply the pattern operator *Bridge* in order to decouple the platform-specific functionality from the domain-specific parts. Further requirements were that the reorganisation must preserve the needed inheritance relationships of both `Graphic` and `Shape`, since some methods of those classes were implemented using inherited functionalities[16] and the process should provide the possibility for appropriate parameterisation by the user (e.g., the selection of delegated methods or inheritance relations).

[16]Note: These inheritance relations were left out for simplicity in Figure 2.55 and 2.56

In order to achieve the necessary target structure, we used a variant of pattern operator *Bridge* consisting of the refactoring `convert_inheritance_into_aggregation`. This high-level refactoring converts an inheritance relationship into a component relationship [JOHN 93] and consists of the following steps:

1. Create a new member variable `impl` of type `Graphic`.
2. For each member function inherited from `Graphic` create a member function in `Shape`. Calls to these functions are delegated to the corresponding methods in `Graphic`.

Due to the requirement, which states that the inheritance relationship should be preserved, the refactoring was slightly modified as follows. Instead of breaking up the inheritance relationship the refactoring `change_superclass` is called for class `Shape`. `Shape` becomes a subclass of the superclass of `Graphic`. Now one can easily change the code for platform-dependent graphic support by creating the appropriate subclasses `GraphicGDI` and `GraphicDDraw`. The corresponding methods of `Graphic` are changed to become *abstract*. All these steps can be supported by tool-based refactorings as well.

The process of applying design pattern operators to existing systems resulted in some difficulties. First, pattern operators must obey existing inheritance relationships if necessary. This sometimes leads to slightly complicated refactoring sequences and often requires user interaction. Another problem was that sometimes our tool could not check all pre-conditions automatically. However, this does not occur in only this specific case but is generally a problem [JOHN 93], since pre-condition checking often requires global data flow analysis. Beyond this, our tool only supported a subset of C++ syntax (not supported is pointer arithmetic and multiple inheritance to name a few). Nevertheless, our tool significantly eased the reorganisation of the software. In addition, our tool works directly on the source code and no pretty printing is required. Thus programmers recognise their own code after the reorganisation operation has been completed.

2.4.4 Related Work

Since reengineering and reorganisation has been recognised as an important field of research, many contributions to this topic can be found in the literature. We present in the following only work which is closely related to our work.

Schema evolution of object-oriented databases can be seen as the basis for Opdykes work. In [BANE 87a] a set of primitive schema evolution operators for the object-oriented database-system *ORION* is presented and categorised. However the focus of this work is more on how to deal with different versions of persistent objects than on code transformations in general.

One aspect of improving the structure of object-oriented systems is discussed in [CASA 92] and [CASA 93]. The author describes both a global and an incremental approach for reorganisation of inheritance relationships. The approach focuses on the reduction of redundancies in the class definitions. The same author presented in [CASA 95b] an excellent overview of existing techniques for the management of class evolution in object-oriented systems.

Several approaches cover parts of the reengineering life-cycle. In [MEIJ 96][VAN 96][FLOR 97] the authors present an approach for visualising *fragments* of object-oriented systems. These fragments can be classes, subsystems or even design patterns. Transformations of the fragments are implemented by using the refactoring browser for Smalltalk [ROBE 97b]. However the authors do not describe any high-level transformations of fragments.

The *MeTHOOD*-project [GROT 97] covers almost the whole reengineering life-cycle. The description of the transformations is given in a way similar to the description of the design pattern operators. However no proofs of correctness are given and it seems that it is not possible to perform the transformations automatically.

Adaptive Programming [LIEB 95][HÜ 96] separates the structure of object-oriented systems from the algorithms operating on these structures. This approach aims at a change avoidance rather than at a systematic way of increasing the flexibility of object-oriented systems.

2.4.5 Summary

In this chapter we presented a new approach to the tool-based reorganisation of object-oriented software. Implementing design pattern operators with refactorings, we combined two existing approaches in order to overcome the weaknesses of both of them. Our methodology uses refactorings, which are proven to preserve behaviour and can be implemented in tools. Since every application of a refactoring leads to a behaviour preserving state, composing design pattern operators of refactorings ensures the behaviour preservingness of the entire pattern operator. Beyond this, design pattern operators are language independent, at least at design level. Pattern operators aim at the right level of abstraction, since most of the problems of an object-oriented system can only be solved by improving the systems design.We demonstrated the feasibility of our approach by applying design pattern operators composed of simple or complex refactorings to an existing system in order to reorganise the hot-spots of that system.

2.5 Reverse and Reengineering Patterns

We have shown how metrics, graphs and metrics combination and grouping software entities can help in understanding identifying important entities, overall applications organizations and detecting problems. Based on these results, refactorings can help fixing the identified problems.

Having techniques for helping in the reengineering tasks is of primary importance. Without techniques and tools supporting them reengineering applications would be nearly impossible. However, relying only on techniques and tools does not suffice. Indeed knowing how to apply a given technique is not enough and knowing when to apply or not is also a key point in the success of the reengineering. Moreover, knowing what are consequence of applying a given technique and in which order apply them is also a key information.

The following part is a first attempt to record all the implicit knowledge that reengineers have been developed over the years. This knowledge is described in pattern format to ease the readability and highlight the motivation and the forces behind every patterns. Please notice that this work is not into a final and definitive stage as the gathering of such an information is slow and difficult.

Part II

Reverse Engineering

Chapter 3

Reverse Engineering Patterns

3.1 Patterns for Reverse Engineering

This pattern language describes how to reverse engineer an object-oriented software system. Reverse engineering might seem a bit strange in the context of object-oriented development, as this term is usually associated with "legacy" systems written in languages like COBOL and Fortran. Yet, reverse engineering is very relevant in the context of object-oriented development as well, because the only way to achieve a truly reusable object-oriented design is recognized to be iterative development (see [BOOC 94], [GOLD 95], [JACO 97], [REEN 96]). Iterative development involves refactoring existing designs and consequently, reverse engineering is an essential facet of any object-oriented development process.

The patterns have been developed and applied during the FAMOOS project [http://www.iam.unibe.ch/~famoos/]; a project whose goal is to produce a set of re-engineering techniques and tools to support the development of object-oriented frameworks. Many if not all of the patterns have been applied on software systems provided by the industrial partners in the project (i.e., Nokia and Daimler-Chrysler). These systems ranged from 50.000 lines of C++ up until 2,5 million lines of Ada. Where appropriate, we refer to other known uses we were aware of while writing.

In its current state, the pattern language can still be improved and we welcome all kinds of feedback that would help us do that. We are especially interested in course grained comments —does the structure work? is the set of forces complete? is the naming OK ?— rather than detailed comments on punctuation, spelling and lay-out.

Acknowledgments. We would like to thank our EuroPLoP'99 shepherd Kyle Brown: his comments were so good we considered including him as a co-author. We also want to thank both Kent Beck and Charles Weir who shepherded a very rough draft of what you hold right now. Finally, we must thank all participants of the FAMOOS project for providing such fruitful working context.

3.2 Clusters of Patterns

The pattern language itself has been divided into *clusters* where each cluster groups a number of patterns addressing a similar reverse engineering situation. The clusters correspond roughly to the different phases one encounters when reverse engineering a large software system. Figure 3.1 provides a road map and below you will find a short description for each of the clusters.

First Contact (p. 109) This cluster groups patterns telling you what to do when you have your very first contact with the software system.

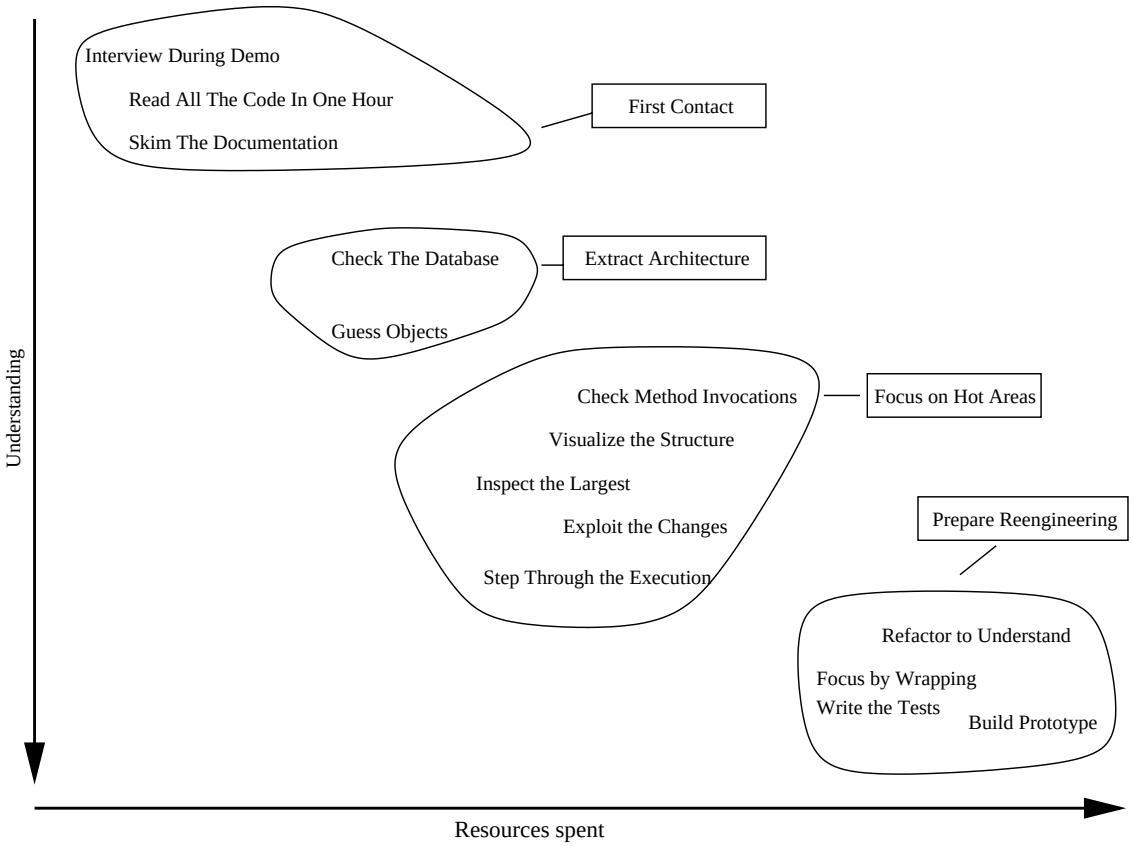


Figure 3.1: Overview of the pattern language using clusters.

Extract Architecture (p. 121) Here, the patterns tell you how to get to the architecture out of a system. This knowledge will serve as a blueprint for the rest of the reverse engineering project.

Focus on Hot Areas (p. 129) The patterns in this cluster describe how to get a detailed understanding of a particular component in your software system.

Prepare Reengineering (p. 145) Since reverse engineering often goes together with reengineering, this cluster includes some patterns that help you prepare subsequent reengineering steps.

3.3 Overview of Forces

All the patterns in this pattern language tell you how to address a typical reverse engineering problem. To evaluate the situation before and after applying the pattern we introduce a number of *forces*. The forces are meant to help you assessing whether the pattern is appropriate for your particular situation.

Limited Resources. Because your resources are limited you must select which parts of the system to reverse engineer first. However, if you select the wrong parts, you will have wasted some of your precious resources. In general the *less resources you need to apply, the better*.

Tools and Techniques. For reverse engineering large scale systems, you need to apply techniques probably accompanied with tools. However, techniques and tools shape your thoughts and good reverse engineering, requires an unbiased opinion. Also, techniques and tools do require resources which you might not be willing to spend. In general, the *less techniques and tools required, the better.*

Reliable Info. A reverse engineer is much like a detective that solves a mystery from the scarce clues that are available [WILL 96b]. As with all good detective stories, the different clues and testimonies contradict each other, thus your challenge is to assess which information is reliable and solve the mystery by coming up with the most plausible scenario. In general, the *more reliable the information you get, the better*.

Abstraction. The whole idea of understanding the inner complexities of a software system is to construct mental models of portions of it, thus a process of abstraction. Consequently, the reengineering taxonomy of Chikofsky and Cross [CHIK 90], defines reverse engineering as "the process of analyzing a subject system to [...] create representations of the system [...] at a higher level of abstraction". Of course, the target level of abstraction for your particular reverse engineering step depends very much on the subsequent demands and so you don't want to get too abstract. Still in general, the *more abstract the information obtained, the better*.

Sceptic Colleagues. As a reverse engineer, you must deal with three kinds of colleagues. The first category are the faithful, the people who believe that reverse engineering is necessary and who thrust that you are able to do it. The second is the category of sceptic, who believe this reverse engineering of yours is just a waste of time and that its better to start the whole project from scratch. The third category is the category of fence sitters, who do not have a strong opinion on whether this reverse engineering will pay off, so they just wait and see what happens. To save your reverse engineering from ending up in the waste bag, you must keep convincing the faithful, gain credit with the fence sitters and be wary of the sceptic. In general, the *more credibility you gain, the better*.

3.4 Resolution of Forces

Table 3.1 shows an overview of how the different patterns resolve the forces. This view is especially important because it emphasises the different trade-offs implied by the patterns. For instance, it shows that READ ALL THE CODE IN ONE HOUR and SKIM THE DOCUMENTATION take about the same amount of resources and also require about the same amount of techniques and tools (very little, hence the double plusses), yet score differently on the reliability and abstraction level of the resulting information. On the other hand, GUESS OBJECTS requires more resources, techniques and tools then the previous two (one minus), but achieves better results in terms of reliable and abstract information.

3.5 Format of a Reverse Engineering Pattern

The pattern presented hereafter have the following format.

Name. Names the pattern after the solution it proposes. The pattern names are verb phrases to stress the action implied in applying them.

Intent. Summarizes the purpose of the pattern, including a clarifying *Example* on when and how to apply the pattern.

Context. Presents the context in which the pattern is supposed to be applied. You may read this section as the prerequisites that should be satisfied before applying the pattern.

Problem. Describes the problem the pattern is supposed to solve. Note that the prerequisites defined in the 'Context' section are supposed to narrow the scope of the problem, so readers are encouraged to read both sections together.

Solution. Proposes a solution to the problem that is applicable in the given context. This section may include a *Recipe* or a list of *Hints* and *Variations* to be taken in account when applying the solution.

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
FIRST CONTACT					
READ ALL THE CODE IN ONE HOUR	++	++	+	-	+
SKIM THE DOCUMENTATION	++	++	-	+	-
INTERVIEW DURING DEMO	++	+	0	+	-
EXTRACT ARCHITECTURE					
GUESS OBJECTS	-	-	++	++	+
CHECK THE DATABASE	-	-	++	+	++
FOCUS ON HOT AREAS					
INSPECT THE LARGEST	0	-	0	0	0
VISUALIZE THE STRUCTURE	-	- -	+	+	+
CHECK METHOD INVOCATIONS	-	-	+	+	0
EXPLOIT THE CHANGES	- -	- -	++	+	++
STEP THROUGH THE EXECUTION	-	0	++	-	+
PREPARE REENGINEERING					
WRITE THE TESTS	- -	-	++	++	0
REFACTOR TO UNDERSTAND	- -	-	0	+	0
BUILD A PROTOTYPE	- -	-	+	- -	++
FOCUS BY WRAPPING	- -	0	0	0	0

Table 3.1: How each pattern resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -. Limited Resources: The less resources you need to apply, the better. Tools and Techniques: The less techniques and tools required, the better. Reliable Info: The more reliable the information you get, the better. Abstraction: The more abstract the information obtained, the better. Sceptic Colleagues: The more credibility you gain, the better.

Forces Resolved. Describes the situation after applying the pattern. This description is done in terms of the forces.

Rationale. Explains the technical background of the pattern, i.e. why it works.

Known Uses. Presents the know uses of this pattern. Note that all patterns in this pattern language have been developed and applied in the context of the FAMOOS project. Yet, this section presents other reported uses of the pattern we were aware of while writing the pattern.

Related Patterns. Links the pattern in a web of other patterns, explaining how the patterns work together to achieve the global goal of reverse engineering. The section includes a *Resulting Context* which tells you how you may use the output of this pattern as input for another one.

Chapter 4

Cluster: First Contact

All the reverse engineering patterns in this cluster are applicable in the very early stage of a reverse engineering project when you are largely unfamiliar with the software system. Before tackling such a project, you need an initial assessment of the software system. However, accomplishing a good initial assessment is difficult because you need a quick and accurate result.

The patterns in this cluster tell you how to optimally exploit information resources like source code (READ ALL THE CODE IN ONE HOUR (p. 111)), documentation (SKIM THE DOCUMENTATION (p. 114)) and system experts (INTERVIEW DURING DEMO (p. 117)). The order in which you apply them depends mainly on your project and we refer you to the *"Related Patterns"* section in each pattern for a discussion on the trade-offs involved. Afterwards you will probably want to CONFER WITH COLLEAGUES (p. 152) and then proceed with EXTRACT ARCHITECTURE (p. 121).

Forces Revisited

Limited Resources. Wasting time early on in the project has severe consequences later on. *Consequently, time is the most precious resource in the beginning of a reverse engineering project.* This is especially relevant because in the beginning of a project you feel a bit uncertain and then it is tempting to start an activity that will keep you busy for a while, instead of something that confronts you immediately with the problems to address.

Tools and Techniques. In the beginning of a reverse engineering project, you are in a bootstrapping situation: you must decide which techniques and tools to apply but you lack a profound basis to make such a decision. *Consequently, choose very simple techniques and very basic tools, deferring complex but time consuming activities until later.*

Reliable Info. Because you are unfamiliar with the system, it is difficult to assess which information is reliable. *Consequently, base your opinion on certified information but complement it using supplementary but less reliable information sources.*

Abstraction. At the beginning of the project you can not afford to be overwhelmed by too many details. *Consequently, favor techniques and tools that provide you with a general overview.*

Sceptic Colleagues. This force is often reinforced in the beginning of a reverse engineering project, because as a reverse engineer –or worse, a consultant– there is a good chance that you are a newcomer in a project team. *Consequently, pay attention to the way you communicate with your colleagues as the first impression will have dire consequences later.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
READ ALL THE CODE IN ONE HOUR	++	++	+	-	+
SKIM THE DOCUMENTATION	++	++	-	+	-
INTERVIEW DURING DEMO	++	+	0	+	-

Table 4.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

READ ALL THE CODE IN ONE HOUR

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Make an initial evaluation of the condition of a software system by walking through its source code in a limited amount of time.

> ***Example.*** *You are facing a 500 K lines C++ program, implementing a software system to display multi-media information in real time. Your boss asks you to look how much of the source code can be resurrected for another project. Before actually identifying what may be reused, you will leaf through the source code to get a feeling for its condition.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have the source code at your disposal and you have reasonable expertise with the implementation language being used.

Problem

You need an initial assessment of the internal state a software system to plan further reverse engineering efforts.

Solution

Take the source code to a room where you can work undisturbed (no telephones, no noisy colleagues). Grant yourself a reasonably short amount of study time (i.e., approximately one hour) to walk through the source code. Take notes sparingly to maximize the contact with the code.

After this reading time, take about the same time to produce a report about your findings, including list of (i) the important entities (i.e., classes, packages, ...); (ii) the coding idioms applied (i.e., C++ [COPL 92], [MEYE 98], [MEYE 96]; Smalltalk [BECK 97]); and (iii) the suspicious coding styles discovered (i.e., "code smells" [FOWL 99]). Keep this report short, and name the entities like they are mentioned in the source code.

Hints. The fact that you are limited in time should force you to think how you can extract the most useful information. Below are some hints for things to look out for.

- Functional tests or units tests convey important information about the functionality of a software system.
- Abstract classes and methods reveal design intentions.
- Classes high in the hierarchy often define domain abstractions; their subclasses introduce variations on a theme.

- Occurrences of the Singleton pattern [GAMM 95] may represent information that is constant for every complete execution of a system.

- Surprisingly large constructs often specify important chunks of functionality that should be executed sequentially.

- Some development teams apply coding styles and if they did, it is good to be aware of them. Especially naming conventions are crucial to scan code quickly.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to collect the source code) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. Good source code browsers can speed you up and inheritance hierarchy browsers can give you a feel for the structure of a software system. However, be wary of fancy tools as they quickly overwhelm you with too much unnecessary information and may require a lot of time to configure correctly. Printing out the source code and reading a paper version may serve just as well.

Reliable Info. The concentrated contact with the code –and code is the only testimony you are sure is correct[1]– provides you with a rather unbiased view to start with. Moreover, by applying this pattern –especially in combination with SKIM THE DOCUMENTATION (p. 114)– you may already have encountered some contradicting pieces of information, which is definitely worthwhile to explore in further depth.

Abstraction. The information you get out is fairly close to the source code, consequently the abstraction level is quite low. However the fact that you work under time pressure forces you to skip details driving you towards an abstract view of the software system.

Sceptic Colleagues. The mere effect of asking quite precise questions after only 1/2 a day of effort raises your credit tremendously, usually enough for being allowed to continue your attempts.

Rationale

Reading the code in a short amount of time is very efficient as a starter. Indeed, by limiting the time and yet forcing yourself to look at all the code, you mainly use your brain and coding expertise to filter out what seems important. This is a lot more efficient than extracting human readable representations or organizing a meeting with all the programmers involved.

Moreover, by reading the code directly you get an unbiased view of the software system including a sense for the details and a glimpse on the kind of problems you are facing. Because the source code describes the functionality of the system –no more, no less– it is the only reliable source of information. Be careful though with comments in the code. Comment can help you in understanding what a piece of software is supposed to do. However, just like other kinds of documentation, comments can be outdated, obsolete or simply wrong.

Finally, acquiring the vocabulary used inside the software system is essential to understand it and communicate about it with other developers. This pattern helps to acquire such a vocabulary.

[1] Remember the old Swiss saying: ”If the map and the terrain disagree, trust the terrain”

Known Uses

While writing this pattern, one of our team members applied it to reverse engineer the Refactoring Browser [ROBE 97a]. The person was not familiar with Smalltalk, yet was able to identify code smells such as "Large Constructs" and "Duplicated Code". Even without Smalltalk experience it was possible to get a feel for the system structure by a mere inspection of class interfaces. Also, a special hierarchy browser did help to identify some of the main classes and the comments provided some useful hints to what parts of the code were supposed to do. Applying the pattern took a bit more than an hour, which seemed enough for a relatively small system and slow progress due to the unfamiliarity with Smalltalk.

The original pattern was suggested by Kent Beck, who stated that it is one of the techniques he always applies when starting consultancy on an existing project. Since then, other people have acknowledged that it is one of their common practices.

Related Patterns

If possible, READ ALL THE CODE IN ONE HOUR (p. 111) in conjunction with SKIM THE DOCUMENTATION (p. 114) to maximize your chances of getting a coherent view of the system. To guide your reading, you may precede this pattern with INTERVIEW DURING DEMO (p. 117), but then you should be aware that this will bias your opinion.

Resulting Context. This pattern results in a list of (i) the important entities (i.e., classes, packages, ...); (ii) the presence of standard coding idioms and (iii) the suspicious coding styles discovered. This is enough to start GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to improve the list of important entities. Depending on whether you want to wait for the results of SKIM THE DOCUMENTATION (p. 114), you should consider to CONFER WITH COLLEAGUES (p. 152).

SKIM THE DOCUMENTATION

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Make an initial guess at the functionality of a software system by reading its documentation in a limited amount of time.

> ***Example.*** *You must develop a geographical information system. Your company has once been involved in a similar project, and your boss asks you to check if some of the design of this previous project can be reused. Before doing any design extraction on the source code, you will skim the documentation to see how close this other system is to what you are expected to deliver.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have the documentation at your disposal and you are able to interpret the diagrams and formal specifications contained within.

> ***Example.*** *If the documentation relies on use cases (see [*JACO *97]) for recording scenarios or formal languages for describing protocols, you should be able to understand the implications of such specifications.*

Problem

You need an initial idea of the functionality provided by the software system in order to plan further reverse engineering efforts.

Solution

Take the documentation to a room where you can work undisturbed (no telephones, no noisy colleagues). Grant yourself a reasonably short amount of study time (i.e., approximately one hour) to scan through the documentation. Take notes sparingly to maximize the contact with the documentation.

After this reading time, take about the same time to produce a report about your findings, including a list of (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) references to relevant design information. Include your opinion on how reliable and useful each of these are. Keep this report as short as possible and avoid redundancy at all cost (among others, use references to sections and/or page numbers in the documentation).

Depending on the goal of the reverse engineering project and the kind of documentation you have at your disposal, you may steer the reading process to match your main interest. For instance, if you want insight into the original system requirements then you should look inside the analysis documentation, while knowledge about which features are actually implemented should be collected from the end-user manual or tutorial notes. If you have the luxury of choice, avoid spending too much time to understand the design documentation (i.e., class diagrams, database schema's, ...): rather record the presence and reliability of such documents as this will be of great help in later stages of the reverse engineering.

Be aware for documentation that is outdated with respect to the actual system. Always compare version dates with the date of delivery of the system and make note of those parts that you suspect unreliable.

Avoid to read the documentation electronically if you are not sure to gain significant browsing functionality (e.g., hypertext links in HTML or PDF). This way you will not spend times with versions, file format and platform issues that certain word processors and CASE tools do not succeed to address.

Hints. The fact that you are limited in time should force you to think how you can extract the most useful information. Below are some hints for things to look out for.

- A table of contents gives you a quick overview of the structure and the information presented.
- Version numbers and dates tell you how up to date the documentation is.
- References to other parts of the documentation convey chronological dependencies.
- Figures are a always a good means to communicate information. A list of figures, if present, may provide a quick path through the documentation.
- Screen-dumps, sample print-outs, sample reports, command descriptions, reveal a lot about the functionality provided in the system.
- Formal specifications, if present, usually correspond with crucial functionality.
- An index, if present contains the terms the author deems significant.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to collect the documentation) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. As reading the documentation only requires the physical document, the tool interference is really low. Yet, when CASE tools have been applied, it may be necessary to consult the documentation on line. Note that CASE tools often enforce some documentation conventions so be sure to be aware of them.

No special techniques are necessary to apply this pattern, unless formal specification or special diagrams are used.

Reliable Info. The success of this pattern depends heavily on the quality of the documentation. Applying this pattern (especially combined with READ ALL THE CODE IN ONE HOUR (p. 111)), you may have encountered some contradicting pieces of information, which is definitely worthwhile to explore in further depth.

Abstraction. The abstraction level you get out depends largely on the abstraction level of the available documentation, but is usually quite high because documentation is supposed to be written at a certain abstraction level.

Sceptic Colleagues. Unless good documentation is available, sceptics will almost certainly consider this activity a waste of time and you will probably loose some credibility with the faithful and fence sitters. This is a negative effect, so reduce its potential impact by limiting the time spend here.

Rationale

Knowing what functionality is provided by the system is essential for reverse engineering. Documentation provides an excellent means to get an external description of this functionality.

However, documentation is either written before or after implementation, thus likely to be out of sync with respect to the actual software system. Therefore, it is necessary to record the reliability. Moreover, documentation comes in different kinds, i.e. requirement documents, technical documentation, end-user manuals, tutorial notes. Depending on the goal of your reengineering project, you will record the usability of each of these documents. Finally, documentation may contain large volumes of information thus reading is time consuming. By limiting the time you spend on it, you force yourself to classify the pieces of information into the essential and the less important.

Related Patterns

You may or may not want to SKIM THE DOCUMENTATION (p. 114) before READ ALL THE CODE IN ONE HOUR (p. 111) depending on whether you want to keep your mind free or whether you want some subjective input before reading the code. INTERVIEW DURING DEMO (p. 117) can help you to collect a list of entities you want to read about in the documentation.

Resulting Context. This pattern results in a list of (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) references to relevant design information plus an opinion on how reliable and useful each of these are. Together with the result of READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114) this is a good basis to CONFER WITH COLLEAGUES (p. 152) and then proceed with GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126).

INTERVIEW DURING DEMO

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Obtain an initial feeling for the functionality of a software system by seeing a demo and interviewing the person giving the demo.

> ***Example.*** *You are asked to extend an existing database application so that it is now accessible via the world-wide web. To understand how the end-users interact with the application, you will ask one of the current users to show you the application and use that opportunity to chat about the systems user-interface. And to understand some of the technical constraints, you will also ask one of the system maintainers to give you a demo and discuss about the application architecture.*

Context

You are starting a reverse engineering project of a large and unfamiliar software system. You have found somebody to demonstrate the system and explain its usage.

Problem

You need an idea of the typical usage scenario's plus the main features of a software system in order to plan further reverse engineering efforts.

Solution

Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Note that the interviewing part is at least as enlightening as the demo.

After this demo, take about the same time to produce a report about your findings, including (i) some typical usage scenarios or use cases; (ii) the main features offered by the system and whether they are appreciated or not; (iii) the system components and their responsibilities; (iv) bizarre anecdotes that reveal the folklore around using the system.

Hints. The person who is giving the demo is crucial to the outcome of this pattern so care is demanded when selecting the person. Therefore, consider to apply this pattern several times with different kinds of persons giving the demo. This way you will see variances in what people find important and you will hear different opinions about the value of the software system. Always be wary of enthusiastic supporters or fervent opponents: although they will certainly provide relevant information, you must spend extra time to look for complementary opinions in order to avoid prejudices.

Below are some hints concerning people you should be looking for, what kind of information you may expect from them and what kind of questions you should ask them.

- An *end-user* should tell you how the system looks like from the outside and explain you some detailed usage scenarios based on the daily working practices. Ask about the situation in the company

before the software system was introduced to assess the scope of the software system within the business processes. Probe for the relationship with the computer department to divulge bizarre anecdotes.

- A person from the *maintenance/development team* should clarify the main requirements and architecture of a system. Inquire how the system has evolved since delivery to reveal some of the knowledge that is passed on orally between the maintainers. Ask for samples of bug reports and change requests to assess the thoroughness of the maintenance process.

- A *manager* should inform you how the system fits within the rest of the business domain. Ask about the business processes around the system to check for unspoken motives concerning your reverse engineering project. This is important as reverse engineering is rarely a goal in itself, it is just a means to achieve another goal.

Forces Resolved

Limited Resources. By applying this pattern, you spend 1/2 a day (plus the time to set-up the demo) to end up with a short list that is a reasonable basis for planning further reengineering efforts.

Tools and Techniques. Except for the equipment necessary to run the software system –which should be readily available– this pattern does not require anything special. The interviewing technique to apply requires a special listening ear though.

Reliable Info. A demo is a reliable means to dig out what features are considered important, but you cannot trust on it to omit irrelevant features. Of course the reliability of the information obtained depends largely on the person who is giving the demo. Therefore, if possibly cross-check any information against other more reliable sources (requirements, progress and delivery reports, source code, log files, ...).

Abstraction. The abstraction level achieved by seeing a demo is quite abstract, though it depends on the person who is giving a demo.

Sceptic Colleagues. The users and maintainers of a software system are usually quite eager to show you the system and tell you what they like and dislike about it. If you have a good listening ear this is a good way to boost your credibility.

Rationale

Interviewing people working with a software system is essential to get a handle on the important functionality and the typical usage scenario's. However, asking predefined questions does not work, because in the initial phases of reverse engineering you do not know what to ask. Merely asking what people like about a system will result in vague or meaningless answers. On top of that, you risk getting a very negative picture because people have a tendency to complain.

Therefore, hand over the initiative to the user by requesting for a demo. First of all, a demo allows users to tell the story in their own words, yet is comprehensible for you because the demo imposes some kind of tangible structure. Second, because users must start from a running system, they will adopt a more positive attitude explaining you what works. Finally, during the course of the demo, you can ask lots of precise questions, getting lots of precise answers, this way digging out the expert knowledge about the system's usage.

Known Uses

One anecdote from the very beginning of the FAMOOS project provides a very good example for the potential of this pattern. For one of the case studies —a typical example of a 3-tiered application with a database layer, domain objects layer and user-interface layer— we were asked 'to get the business objects out'. Two separate individuals were set to that task, one took a source code browser and a CASE tool and extracted some class diagrams that represented those business objects. The other installed the system on his local PC and spend about an hour playing around with the user interface to came up with a list of ten questions about some strange observations he made. Afterwards, a meeting was organized with the chief analyst-designer of the system and the two individuals that tried to reverse engineer the system. When the analyst-designer was confronted with the class-diagrams he confirmed that these covered part of his design, but he couldn't tell us what was missing nor did he tell us anything about the rationale behind his design. It was only when we asked him the ten questions that he launched off into a very enthusiastic and very detailed explanation of the problems he was facing during the design — he even pointed to our class diagrams during his story! After having listened to the analyst-designer, the first reaction of the person that extracted the class diagrams from the source code was 'Gee, I never read that in the source code'.

Related Patterns

For optimum results, you should perform several attempts of INTERVIEW DURING DEMO (p. 117) with different kinds of people. Depending on your taste, you may perform these attempts before, after or interwoven with READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114).

Resulting Context. This pattern results in (i) some typical usage scenarios or use cases; (ii) the main features offered by the system and whether they are appreciated or not; (iii) the system components and their responsibilities; (iv) bizarre anecdotes that reveal the folklore around using the system. Together with the result of READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114) this is a good basis to CONFER WITH COLLEAGUES (p. 152) and then move on to GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126).

Chapter 5

Cluster: Extract Architecture

The patterns in FIRST CONTACT (p. 109) should have helped you getting an initial feeling of the software system. Now is the right time to draw some blueprints of the complete system that will serve as a roadmap during the rest of the reverse engineering project. The main priority in this stage of reverse engineering is to get an accurate picture without spending too much time on the hairy details.

The patterns in this cluster tell you how to derive a system blueprint from source code (GUESS OBJECTS (p. 123)) and from a database schema (CHECK THE DATABASE (p. 126)). With these blueprints you will probably want to proceed with FOCUS ON HOT AREAS (p. 129).

Forces Revisited

Reliable Info. Since the blueprints resulting from these activities will influence the rest of your project, accuracy is the single most important aspect. *Consequently, take special precautions to make the extracted blueprints as reliable as possible.* In particular, plan for an incremental approach where you gradually improve the blueprints while you gain a better understanding of the system.

Limited Resources. Results coming from this stage of reverse engineering are always worthwhile. *Consequently, consider* EXTRACT ARCHITECTURE *a very important activity and plan to spend a considerable amount of your resources here.* However, via an incremental approach you can stretch your resources in time.

Tools and Techniques. While extracting an architecture, you can afford the time and money to apply some heavyweight techniques and purchase some expensive tools. *Yet —because accuracy is so important— do never rely on techniques and tools and always make a conscious assessment of their output.*

Abstraction. Architectural blueprints are meant to strip away the details. Yet, computer science has this strange phenomenon that details are crucial to the overall system [BROO 87]. *Consequently, favor different blueprints that emphasize one perspective and choose the most appropriate ones when necessary.* Adapt the notation to the kind of blueprint you are making ([DAVI 95] – principle 21).

Sceptic Colleagues. Good blueprints help a lot because they greatly improve the communication within a team. However, since they strip away details, you risk to offend those people who spend their time on these details. Also, certain notations and diagrams may be new to people, and then your diagrams will just be ignored. *Consequently, take care in choosing which blueprints to produce and which notations to use — they should be helpful to all members of the team.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
GUESS OBJECTS	-	-	++	++	+
CHECK THE DATABASE	-	-	++	+	++

Table 5.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

GUESS OBJECTS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Progressively refine a model of a software system, by defining hypotheses about what should be in the code and checking these hypotheses against the source code

> ***Example.*** *You are facing a 500 K lines C++ program, implementing a software system to display multi-media information in real time. Your boss asks you to look how much of the source code can be resurrected from another project. After having* READ ALL THE CODE IN ONE HOUR *(p. 111), you noticed an interesting piece of code concerning the reading of the signals on the external video channel. You suspect that the original software designers have applied some form of observer pattern, and you want to learn more about the way the observer is notified of events. You will gradually refine your assumption that the class* VIDEOCHANNEL *is the subject being observed by reading its source code and tracing interesting paths.*

Context

You are in the early stages of reverse engineering a software system: you have an initial understanding of its functionality and you are somewhat familiar with the main structure of its source code. Due to this, you have identified a certain aspect of the system as especially important. You have on-line access to the source code of the software system and the necessary tools to manipulate it (i.e., from an elementary grep to a professional browser). You have reasonable expertise with the implementation language being used.

Problem

You must gain an overall understanding of the internal structure of a software system and report this knowledge to your colleagues so that they will use it as a kind of roadmap for later activities.

Solution

Take a notepad and/or sketchpad (not necessarily as an electronic tool). Based on your experience, and the little you already understand from the system, devise a model that serves as your initial hypotheses of what to expect in the source code. Check these hypotheses against the source code, using whatever tools you have available. Consciously keep track of which parts of the source code confirm and which parts contradict your hypotheses. Based on the latter, refine the initial model, recheck the hypotheses and rework the list of confirmations and contradictions. Do this until you obtain a more or less stable model.

Note that it is a good idea to sort the entities in your hypotheses models according to the probability of appearance in source-code. This is especially useful as names inside the source-code do not always match with the concepts they represent. This may be due to particular coding conventions or compiler restrictions (identifiers cannot exceed a certain length), or because of the native language of the original programmer.[1]

Afterwards, sit down to produce a boxes- and arrows diagram describing your findings. As a rule of the thumb, make sure your diagram fits on one page. It is better to have two distinct diagrams, where each

[1] In one particular reverse engineering experience, we were facing source code that was a mixture of English and German. As you can imagine, grep is not a very good tool to check occurrences of English terms in German texts.

provides a clean perspective on the system than one messy diagram with too many details too read and memorize. People should be able to redraw the diagram from memory after they have seen it once; it is only then that your diagram will really serve as a roadmap.

Variations. The pattern itself is quite broad and thus widely applicable. Below are some suggestions of possible variants.

- *Guess Patterns.* While having READ ALL THE CODE IN ONE HOUR (p. 111), you might have seen some symptoms of patterns. You can use a variant of GUESS OBJECTS to refine this knowledge. (See the better known pattern catalogues [GAMM 95], [BUSC 96], [FOWL 97b] for patterns to watch out for. See also [BROW 96] for a discussion on tool support for detecting patterns.)
- *Guess Object Responsibilities*. Based on the requirements resulting from SKIM THE DOCUMENTATION (p. 114), you can try to assign object responsibilities and check the resulting design against the source code. (To assign object responsibilities, use the noun phrases in the requirements as the initial objects and the verb phrases as the initial responsibilities. Derive a design by mapping objects on class hierarchies and responsibilities on operations. See [WIRF 90] for an in depth treatment on responsibility-driven design.)
- *Guess Object Roles.* The usage scenarios that you get out of INTERVIEW DURING DEMO (p. 117) may serve to define some use cases that in turn help to find out which objects fulfill which roles. (See [JACO 92] for use cases and [REEN 96] for role modeling.)
- *Guess Process Architecture*. The object-oriented paradigm is often applied in the context of distributed systems with multiple cooperating processes. A variant of GUESS OBJECTS may be applied to infer which processes exist, how they are launched, how they get terminated and how they interact. (See [LEA 96] for some typical patterns and idioms that may be applied in concurrent programming.)

Forces Resolved

Limited Resources. The amount of resources you invest in this pattern depends mainly on the level of detail and accuracy that you want to achieve. Be wary of the hairy details though, as this pattern tends to have an exponential effort/gain curve. For detailed information, consider switching to STEP THROUGH THE EXECUTION (p. 142) instead.

Tools and Techniques. Applying this pattern does not require a lot of tools: a a simple grep may be sufficient and otherwise a good code browser will do. Probably you will also need a tool for producing the final blueprint, as it is likely that someone will have to update the blueprint later on in the project. However, choose a a simple drawing tool rather then a special purpose CASE tool, as you will need a lot of freedom to express what you found.

In itself, the pattern does not require a lot of techniques. However, a large repertoire of knowledge about idioms, patterns, algorithms, techniques is necessary to recognize what you see. As such, the pattern should preferably be applied by experts, yet lots of this expertise may be acquired on the job.

Reliable Info. The blueprints you extract by applying this pattern are quite reliable because of the gradual refinement of the hypotheses and confirmation against source code. Yet, be sure to keep the blueprint up to date while your reverse engineering project progresses and your understanding of the software system grows.

Abstraction. If applied well, the different blueprints you achieve by means of GUESS OBJECTS provide the ideal abstraction level. That is, each blueprint provides a unique perspective on the software system that highlights the important facts and strips the unimportant details. Yet, navigating between the various blueprints provides you all the necessary perspectives to really understand the system.

Sceptic Colleagues. The results of GUESS OBJECTS pattern should drastically increase the confidence of your team in the success of the reverse engineering project. This is because the members of the team will normally experience an "*aha erlebness*", where the little pieces of knowledge they have fit the larger whole.

Rationale

Clear and concise descriptions of a system are a necessary ingredient to plan team activities. However, being clear and concise is for humans to decide, thus creating them requires human efforts. On the other hand, they must accurately reflect what's inside the system, so somehow the source-code should be incorporated in the creation process as well. GUESS OBJECTS addresses this tension by using a mental model (i.e., the hypotheses) as the primary target, yet progressively refines that model by checking it against source code. Moreover, conciseness implies loss of detail, hence the reason to extract multiple blueprints offereing alternative perspectives.

Known Uses

In [MURP 97], there is a report of an experiment where a software engineer at Microsoft applied this pattern (it is called 'the Reflexion Model' in the paper) to reverse engineer the C-code of Microsoft Excel. One of the nice sides of the story is that the software engineer was a newcomer to that part of the system and that his colleagues could not spend too much time to explain him about it. Yet, after a brief discussion he could come up with an initial hypotheses and then use the source code to gradually refine his understanding. Note that the paper also includes a description of a lightweight tool to help specifying the model, the mapping from the model to the source code and the checking of the code against the model.

Related Patterns

All the patterns in the FIRST CONTACT (p. 109) cluster are meant to help you building the initial hypotheses to be refined via GUESS OBJECTS (p. 123). Next, some of the patterns in FOCUS ON HOT AREAS (p. 129) may help you to refine this hypothesis.

Resulting Context. After this pattern, you will have a series of blueprints where each contains one perspective on the whole system. These blueprints will help you during later reverse engineering steps, in particular the ones in FOCUS ON HOT AREAS (p. 129) and PREPARE REENGINEERING (p. 145). Consequently, consider applying CONFER WITH COLLEAGUES (p. 152) after applying GUESS OBJECTS (p. 123).

CHECK THE DATABASE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Get a feeling for the data model inside a software system by checking the database schema.

> ***Example.*** *You are asked to extend an existing database application so that it is now accessible via the world-wide web. The initial software system manipulates the business objects (implemented in C++) stored inside a relational database. You will reconstruct the data model underlying your business objects by mapping the table definitions in the database on the corresponding C++ classes.*

Context

You are in the early stages of reverse engineering a software system, having an initial understanding of its functionality. The software system employs some form of a database to make its data persistent.

You have access to the database and the proper tools to inspect its schema. Or even better, you have samples of data inside that database and maybe you are even able to spy on the database queries during the execution of the system. Finally, you have some expertise with databases and knowledge of how data-structures from your implementation language are mapped onto the data-structures of the underlying database.

Problem

You want to derive a data model for the persistent data in a software system in order to guide further reverse engineering efforts.

Solution

Check the database schema to reconstruct at least the persistent part of the data model. Use your knowledge of how constructs in the implementation language are mapped onto database constructs to reverse engineer the real data model. Make samples of data inside the database to refine the data-model.

Forces Resolved

Limited Resources. Reconstructing the data model from the database schema takes considerable resources, although it depends largely on the underlying technology. Factors that affect this force in a positive way are the quality of the database schema (is it in normal form?), the correspondence between the database paradigm and the implementation language paradigm (inheritance hierarchies do not map directly to relational tables), the expressiveness of the database schema (does it include foreign keys ?). On the other hand, the reverse engineering of database schemas may include techniques like data sampling and run-time inspection, which takes even more resources.

Tools and Techniques. This pattern can do without a lot of tool support: a dump of the database schema and some samples of data inside the tables is something all database systems can provide. However,

there are some tools available to support you in recovering object models (see [HAIN 96], [PREM 94], [JAHN 97]).

This pattern requires substantial technical expertise, because it requires knowledge of ways to manipulate data structures in both the implementation language and the database, plus ways to map one onto the other.

Reliable Info. Because the pattern is based on analyzing persistent data, the reliability of the reconstructed data model is usually quite high. However, if the database system is manipulated by different software systems and if each of these software systems is build with different implementation technologies (CASE tools, 4GL, ...), the reliability of the data model tends to decrease because the database schema provides the most common denominator of all implementation technologies involved. Data sampling is a good way to cope with this problem though.

Abstraction. The abstraction level of the reconstructed data model tends to be low, as it is closer to the underlying database schema than it is to the implementation language. However, this depends largely on the amount of resources spent. For instance, with data sampling and run-time inspection one can drastically improve the abstraction level.

Sceptic Colleagues. If applied well, this pattern increases your credibility considerably, because a well defined data model is normally considered a collective source of knowledge which greatly improves the communication within a team. Moreover, almost all software engineers will have experience with data models and will appreciate their presence.

Rationale

Having a well defined central data model is a common practice in larger software projects that deal with persistent data. Not only, it specifies common rules on how to access certain data structures, it is also a great aid in assigning development tasks. Therefore, it is a good idea to extract an accurate data model before proceeding with other reverse engineering activities.

Known Uses

The reverse engineering and reengineering of database systems is a well-known problem, drawing certain attention in the literature (see [HAIN 96], [PREM 94], [JAHN 97]). Note the recurring remark that the database schema alone is too weak a basis and that data sampling and run-time inspection must be included for successful reconstruction of the data model.

Related Patterns

CHECK THE DATABASE requires an initial understanding of the system functionality, like obtained by applying patterns in the cluster FIRST CONTACT (p. 109).

There are some patterns that describe various ways to map object-oriented data constructs on relational database counterparts. See among others [KELL 98], [COLD 99].

Resulting Context. CHECK THE DATABASE results in a data model for the persistent data in your software system. Such a data model is quite rough, but it may serve as an ideal initial hypotheses to be further refined by applying GUESS OBJECTS (p. 123). The data model should also be used as a collective knowledge that comes in handy when doing further reverse engineering efforts, for instance like in the clusters FOCUS ON HOT AREAS (p. 129) and PREPARE REENGINEERING (p. 145). Consequently, consider to CONFER WITH COLLEAGUES (p. 152) after CHECK THE DATABASE.

Chapter 6

Cluster: Focus on Hot Areas

The patterns in FIRST CONTACT (p. 109) should have helped you getting an initial feeling of the software system, while the ones in EXTRACT ARCHITECTURE (p. 121) should have aided you deriving some blueprints of the overall system structure. The main priority now is to get detailed knowledge about a particular part of the system.

This cluster tell you *how*, and to some degree *where* you might obtain such detailed knowledge. The patterns involve quite a lot of tools and rely on substantial technical knowledge, hence are applicable in the later stages of a reverse engineering project only. Indeed, only then you can afford to spend the resources obtaining detailed information as only then you have the necessary expertise to know that your investment will pay off.

There are two patterns that explain you *where* to focus your attention: INSPECT THE LARGEST (p. 131) suggests to look at large object objects, while EXPLOIT THE CHANGES (p. 135) advises to look at the places where programmers have been changing the system. (Of course, no technique or tool will replace the human mind, hence to know where to focus your attention, be sure to CONFER WITH COLLEAGUES (p. 152) as well). Then, there are two patterns that inform you *where and how* to study program structures: VISUALIZE THE STRUCTURE (p. 138) tells about program visualisation techniques, while CHECK METHOD INVOCATIONS (p. 140) recommends to check invocations of both constructor and overridden methods. Finally, there is one pattern describing you *how* to investigate programs, namely STEP THROUGH THE EXECUTION (p. 142) which explains how to take advantage of your debugger.

Many reverse engineering projects prepare for a subsequent reengineering phase. If you're in such a situation, you might consider the patterns in PREPARE REENGINEERING (p. 145) as your next step. If you're not, then these patterns are the last ones we have to offer for helping you.

Forces Revisited

Tools and Techniques. To obtain the required details from a software system you must pay the price in terms of technical expertise and tools. This is the most important force during this stage of reverse engineering: *consequently, make sure your reverse engineering team does possess the necessary skills and tools.*

Limited Resources. These patterns are applicable during the later stages of a reverse engineering project, thus resources are less scarce as you can be quite sure that your investment will pay off. On the other hand, the activities you apply require more resources. *Consequently, engage in detailed reverse engineering only when you are certain that you need to know the details about that part of a system*. The patterns in the previous clusters should have helped you obtaining that knowledge.

Abstraction. All patterns in this cluster have in common that they extract detailed information, at an intermediate level of abstraction (i.e., between source code and design). Yet, detailed knowledge is necessary because in software engineering —and this is in contrast with many other engineering disciplines— details are very important [BROO 87]. So, even during fine-grained reverse engineering, there are little details that seem so obvious, yet may obstruct the understanding of the system if you failed to state them.[1] *Consequently, when working on intermediate abstraction levels, make sure you provide enough context so that the relationship with both higher and lower levels is clear.*

Reliable Info. As details are so important, you should be confident in the obtained results. *Consequently, favour extracting information from the trustworthy information sources.* Fortunately, because you're in the later stages of reverse engineering, you know which information sources are reliable and which ones are not.

Sceptic Colleagues. You would not have arrived this far without the support of some colleagues, so at least you still have the support of the faithful. Moreover, you probably did satisfy the expectations, otherwise the sceptic would have succeeded to cancel your project. And if you did really well, you might even have won some fence sitters over into the camp of the faithful. At this stage, you will not achieve more support from your colleagues. *Consequently, keep on delivering the necessary results to avoid providing reasons for the sceptics to cancel your project.*

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
INSPECT THE LARGEST	0	-	0	0	0
VISUALIZE THE STRUCTURE	-	- -	+	+	+
CHECK OVERRIDDEN METHODS	-	-	+	+	0
EXPLOIT THE CHANGES	- -	- -	++	+	++
STEP THROUGH THE EXECUTION	-	0	++	-	+

Table 6.1: How each pattern of FIRST CONTACT resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

[1] A typical example of such a harmful detail is the use of private/protected in a UML diagram. Depending on the favourite programming language of the author of the diagram, the interpretation is quite different, and readers of the diagram should be made aware of this. That is, with a C++ background the interpretation is class based, thus instances of the same class may access each other's private members. On the other hand, with a Smalltalk background, the interpretation is instance based, thus it is only the object itself that is allowed to access its members. Finally, in Java a protected member may also be accessed by classes in the same package.

INSPECT THE LARGEST

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Identify important functionality by looking at large constructs.

> ***Example.*** *You are facing an object-oriented system and you want to find out which classes do the bulk of the work. You will produce a list of all classes where the number of methods exceeds the average number of methods per class, sort the list and inspect the largest classes manually.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of its functionality and you know the main structure of its source code. You have a metrics tool at your disposal plus a code browser to inspect the source code. The metrics tool is configured in such a way that it provides you with a number of measurements of source code constructs you are feeding into it. Moreover, the metrics are defined in such a way that they have a high correlation with the amount of functionality implemented in the construct.

Problem

You must identify those places in the source code that correspond with important chunks of functionality.

Solution

Use the metrics tool to collect a limited set of measurements for all the constructs in the system. Sort the resulting list according to these measurements. Browse the source code for the largest among those constructs in order to understand how these constructs work together with other related constructs. Produce a list of all the constructs that appear important, including a description of how they should be used (i.e. external interface).

Hints. Identifying important pieces of functionality in a software system via measurements is a tricky business which requires expertise in both data collection and interpretation. Below are some hints that might help you getting the best out of your data.

- *Which metrics to collect ?* In general, it is better to stick to the simple metrics, as the more complex ones will not perform better for the identification of large constructs. This experience is backed up by empirical evidence, as it has been reported in the literature that size metrics have a high correlation (see among others [FENT 97]).

  For identifying important functionality in object-oriented source code, look at methods and classes. For methods you may restrict yourself to counting the lines of code and if available the number of other methods invoked.[2] For classes, you should count the number of methods and the number of

[2] Counting the lines of code can be done very efficiently without parsing, just by counting all occurrences of the <CR> character.

attributes defined on that class, plus the depth of the inheritance tree and probably also the lines of code (i.e., the sum of all the lines of code of all the methods of a class). (See the chapter on Metrics — p.22 for a more precise definition of each of the metrics and a list of other metrics you might collect).

- *Which variants to use ?* Usually, it does not make a lot of difference which variant is chosen, as long as the choice is clearly stated and applied consistently. Here as well, it is preferable to choose the most simple variant, unless you have a good reason to do otherwise. For instance, while counting the lines of code, you should decide whether to include or exclude comment lines, or whether you count the lines after the source code has been normalised via pretty printing. In such a case, do not exclude comment lines nor normalise the source code as the extra effort will not pay of. Another example of an alternative definition is the case of counting the number of methods, where one must decide how to deal with 'special' methods like class methods (i.e., the C++ static methods). In this case it is a good idea to count class methods separately as they represent a different kind of functionality.

- *Which thresholds to apply ?* It is better not to apply thresholds to filter out those constructs which measurements fall into a given threshold interval. Indeed, 'large' is a relative notion and thresholds will distort your perspective of what constitutes large within the system.

- *How to interpret the results ?* Do not only look for the largest construct while analysing the data. Before actually browsing the source code, check the distribution of measurements to see whether the 80/20 rule is satisfied. Also, gather several measurements in different columns one beside another and then look for unusual rows.

  (Note that the 80/20 rule is a more formal expression of the rule of the thumb that most constructs in source code will be small, and only a few exceptional cases will be large. To be precise, the rule states that 80% of the constructs will be smaller than 20% of the size of the largest construct.)

Forces Resolved

Limited Resources. Once the metric tool is configured for the particular language of the software system, collecting the necessary data is not that resource consuming; in the worst case it can be done via batch jobs during the night. However, analysing the data and browsing the selected source code constructs requires a lot of resources depending on the desired level of detail. You can neutralise this effect to some extent by limiting the set of metrics.

Tools and Techniques. To apply this pattern, you require a tool which should be able to collect the necessary measurements. However, since you can restrict yourself to simple counting metrics such a tool should be quite easy to obtain.

Analysing and interpreting the data however, requires a certain amount of knowledge. Some of this knowledge is summarised in our list of metric definitions (see the chapter on Metrics — p.22) and the rest you can learn on the job.

Reliable Info. It is not because a software construct is large that it is important, neither is it true that a small construct is always irrelevant. Therefore, the results contain quite a lot of noise, hence are somewhat unreliable. Still, given the amount of resources required, this pattern usually provides a good return on investment, especially since the large constructs will often point you to other more important but smaller constructs.

Abstraction. The abstraction level of this pattern results mainly from browsing the source code, not so much from measuring the constructs in the system. Therefore, the abstraction level of the results should be considered quite low.

Sceptic Colleagues. Metrics are often associated with process and quality control, therefore some programmers may believe that you will use the metrics to examine their productivity. Be careful if

you have such programmers among your faithful as it may be a way to turn them into sceptics. In particular, do not blindly deduce that large constructs are bad and should be rewritten.

Rationale

The main reason why size metrics are often applied during reverse engineering is because they provide a good focus (between 10 to 20% of the software constructs) for a low investment, even though the results are somewhat unreliable. With such a good focus, you can afford some erroneous results which you will compensate anyway via code browsing.

The results are a bit unreliable because 'large' is not necessarily the same as 'important'. Quite often large constructs are irrelevant as they would have been refactored into smaller pieces if they were important. Conversely, small constructs may be far more important than the large ones, because good designers tend to distribute important functionality over a number of highly reusable and thus smaller components. Still, different larger constructs may share the same smaller construct, so via the larger constructs you may be able to identify some important smaller constructs too.

The main disadvantage of the pattern is that it forces you to look at the largest constructs first. Large constructs are usually the most complicated ones, therefore understanding the corresponding source code may prove to be difficult. Another disadvantage is that the analysis of the metrics data results in a list of raw software constructs. For program understanding, it is usually more important to know how these constructs work together with other ones, something which must be revealed by code browsing.

Note that by restricting yourself to a limited set of simple metrics you already avoid one of the most common pitfalls. Indeed, metrics tools usually offer you a wide range of metrics and since collecting data is so easy, you may be tempted to apply all metrics that are available. However, the more data you collect, the more data you must analyse and the amount of numbers will quickly overwhelm you. Moreover, some metrics require substantial parsing effort, which in turn requires the configuration of the parser to your software system, which can be painstaking and time-consuming. By limiting the amount a metrics and keeping the metrics simple, you circumvent these problems.

Known Uses

In several places in the literature it is mentioned that looking for large object constructs helps in program understanding (see among others, [MAYR 96a], [KONT 97], [FIOR 98a], [FIOR 98b], [MARI 98], [LEWE 98a], [NESI 88]). Unfortunately, none of these incorporated an experiment to count how much important functionality remains undiscovered. As such it is impossible to assess the reliability of size metrics for reverse engineering.

Note that some metric tools visualise information via typical algorithms for statistical data, such as histograms and Kiviat diagrams. Visualisation may help to analyse the collected data. Datrix [MAYR 96a], TAC++ [FIOR 98a], [FIOR 98b], and Crocodile [LEWE 98a] are tools that exhibit such visualisation features.

Related Patterns

Looking at large constructs requires little preparation but the results are a bit unreliable. By investing more in the preparation you may improve the reliability of the results. For instance, if you VISUALIZE THE STRUCTURE (p. 138) you invest in program visualisation techniques to study more aspects of the system in parallel, thereby increasing the quality of the outcome. Also, you can EXPLOIT THE CHANGES (p. 135) to focus on those parts of the system that change, thereby increasing the likelihood of identifying interesting constructs and focussing on the way constructs work together.

Resulting Context. By applying this pattern, you will have identified some constructs representing important functionality. Some other patterns may help you to further analyse these constructs. For instance, if you VISUALIZE THE STRUCTURE (p. 138) you will obtain other perspectives and probably other insights as well. Also, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the runtime behaviour. Finally, in the case of a object-oriented code, you can CHECK METHOD INVOCATIONS (p. 140) to find out how a class is related to other classes.

Even if the results have to be analysed with care, some of the larger constructs can be candidates for further reengineering: large methods may be split into smaller ones (see [FOWL 99]), just like big classes may be cases of a GOD CLASS (see [BROW 98]).

EXPLOIT THE CHANGES

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Recover design issues by asking where, how and why the developers have been changing the implementation.

> ***Example.*** *You must understand an old but evolving software system, where the evolution is controlled through a configuration management system. You will filter out those modules that have been changed most often and find out what these changes where about and why they were necessary.*

> ***Example.*** *You must understand an object-oriented framework that has been adapted several times as the developers gained insight into the problem domain. You will filter out all classes where the number of methods and attributes has decreased significantly and find out where that functionality has been moved to. With that knowledge, you will make a guess at the design rationale underlying this redistribution of functionality.*

Context

You are in a later stage of reverse engineering an evolving software system. You have an overall understanding of its functionality and you know the main structure of its source code. You have several releases of the source code at your disposal plus a way to detect the differences between the releases, i.e. a configuration management system and/or a metrics tool.

Problem

You must identify those parts in the design that played a key role during the system's evolution.

Solution

Use whatever means at your disposal to compile a list of targets of important/frequent changes. For each target, put yourself in the role of the original developer and ask yourself what the change is about and why it was necessary. With this insight, produce a list of crucial system parts, including a description of the design issues that makes them important.

Variations. The pattern comes in two variants corresponding to the way the targets of changes are identified.

- *The configuration database variant* requires that all changes to the system were done via a configuration management system which logs all changes in the configuration database. In that case you can take advantage of the query facilities provided by the configuration database to produce a list of components that have been changed. Sort the list according to the frequency of changes and inspect the corresponding source code plus the comments in the configuration database to find out how and why this component has changed.

- *The change metrics variant* identifies changes by comparing subsequent releases and measuring differences in size. With the change metrics variant, the first step is to measure the size of named constructs in two subsequent releases. Afterwards, you compile a list with three columns: the name of the construct and both measurements. Sort the list according to the largest decrease in size. For each decrease in size, ask yourself where this functionality has been moved to and then deduce how and why this construct has changed.

Hints. If you consider applying the change metrics variant on object-oriented source code, we can recommend three heuristics that help identifying the following changes.

- *Split into superclass / merge with superclass.* Look for the creation or removal of a superclass (change in hierarchy nesting level - HNL), together with a number of pull-ups or push-downs of methods and attributes (changes in number of methods - NOM and number of attributes - NOA).
- *Split into subclass / merge with subclass.* Look for the creation or removal of a subclass (change in number of immediate subclasses- HNL), together with a number of pull-ups or push-downs of methods and attributes (changes in number of methods - NOM and number of attributes - NOA).
- *Move functionality to superclass, subclass or sibling class.* Look for removal of methods and attributes (decreases in number of methods - NOM and number of attributes - NOA) and use code browsing to identify where this functionality is moved to.
- *Split method / factor out common functionality.* Look for decreases in method size (via lines of code - LOC, or number of message sends - MSG, or number of statements - NOS) and try to identify where that code has been moved to.

Rationale

A configuration management tool maintains and controls the different versions of the components that constitute the entire software system. If such a tool has been used for the software system you are reverse engineering, its database contains a wealth of information about where, how and why the software system has evolved. As a reverse engineer, you should exploit the presence of this database.

But even without a configuration management system, it is feasible to identify where, how and to some degree why a system has evolved by comparing subsequent releases and measuring changes. With change metrics, the results are less accurate than it is the case with the configuration database variant, mainly because the rationale for the change is not recorded thus must be deduced. On the other hand, because you focus on constructs that decrease in size, you are likely to identify places where functionality has been moved to other locations. Such moving of functionality is always relevant for reverse engineering, as it reveals design intentions from the original developers.

Satisfying the prerequisite of having different releases of the source code plus the necessary tools to assess the differences, the main advantages of looking at changes are the following. (i) It concentrates on relevant parts, because the changes point you to those places where the design is expanding or consolidating. (ii) It provides an unbiased view of the system, because you do not have to formulate assumptions of what to expect in the software (this is in contrast to GUESS OBJECTS (p. 123) and VISUALIZE THE STRUCTURE (p. 138)) (iii) It gives an insight in the way components interact, because the changes reveal how functionality is redistributed among constructs (this is in contrast to INSPECT THE LARGEST (p. 131)).

Known Uses

There is a company called MediaGeniX, which incorporates a scaled down version of the configuration database variant into their development process and tools. It is based on the so-called tagging tool, which

automatically updates one comment line in a method body each time this method is modified. The comment line records information like the date of the change, the name of the programmer and a reference into their configuration management system. The reference reveals the nature of the change (i.e., bug fix or a new feature) and via consultation of the actual configuration management system even what this change was about. Afterwards, they run queries to identify which features are localised to a few modules and which features cross-cut a large number of modules to identify where they may improve the design of the framework. Also, they have identified methods that are modified a lot when bug fixing, and used this information as input for their code reviewing. They have even identified cycles in the bug fixing, in the sense that the modification of one method fixed a bug but immediately introduced another bug and then the repair of the newly introduced bug again introduced the older bug. More information about the usage of the tagging tool in the context of reverse engineering can be found in [HOND 98].

Besides the tagging tool, we are aware of two other projects where people have been exploiting the version control system for reverse engineering purposes. First, there is the SeeSoft tool, developed at Bell Labs, which visualises source code changes and has been used successfully for reverse engineering purposes [BALL 96]. Second, there is the ARES project (see http://www.infosys.tuwien.ac.at/Projects/ARES/) which also experimented with visualisation of changes using the 3DSoftVis tool [JAZA 99].

Finally, concerning the change metrics variant, we ran an experiment on three medium sized systems implemented in Smalltalk. As reported in [DEME 99c], these case studies suggest that the heuristics support the reverse engineering process by focussing attention on the relevant parts of a software system.

Related Patterns

Inspecting changes is a costly but very accurate way of identifying areas of interest in a system. If you VISUALIZE THE STRUCTURE (p. 138) or INSPECT THE LARGEST (p. 131) you will get less accurate results for a lower amount of resources.

Resulting Context. By applying this pattern, you will have identified some parts in the design that played a key role during the system's evolution. Some other patterns may help you to further analyse these constructs. For instance, if you VISUALIZE THE STRUCTURE (p. 138) you will obtain other perspectives and probably other insights as well. Also, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour. Finally, in the case of a class, you can CHECK METHOD INVOCATIONS (p. 140) to find out how this class is related to other classes.

VISUALIZE THE STRUCTURE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Obtain insight in the software system's structure —including potential design anomalies— by means of well-known visualisations.

> ***Example.*** *You want to understand an object-oriented class structure in order to improve it. In particular, you would like to redistribute responsibilities, by splitting large superclasses and hooking the subclasses underneath the appropriate ancestor. To analyse the situation, you will display the inheritance hierarchies, paying special attention to large classes high up in the hierarchy. Afterwards, for classes identified that way, you will display a graph showing which method accesses which attributes to analyse the class' cohesion and find out whether a split is feasible.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of the system's functionality and based on that understanding, you have selected part of the software system for further inspection. You have a program visualisation tool at your disposal plus a code browser to inspect the source code.

Problem

You want to obtain insight in the structure of a selected part of a software system, including knowledge about potential design anomalies.

Solution

Instruct the program visualisation tool to show you a series of graphical layouts of the program structure. Based on these graphical layouts, formulate yourself some assumptions and use the code browser to check whether your assumptions are correct. Afterwards, produce a list of correct assumptions, classifying the items in one of two categories: (i) helps program understanding, or (ii) potential design anomaly.

Hints. Obtaining insight in the structure of a software system via visualisation tools is difficult, especially when searching for potential design anomalies. We have included our expertise with program visualisation in a separate chapter and we refer the interested reader to the chapter on Visualisation — p.31 for further details.

Rationale

Program visualisation is often applied in reverse engineering because good visual displays allow the human brain to study multiple aspects of a problem in parallel. This is often phrased as "one picture conveys a

thousand words", but then of course the problem is which words they convey, thus which program visualisations to apply and how to interpret them. For the program visualisations listed in the the chapter on Visualisation — p.31 we describe both when to apply them and how to interpret the results. For other visualisations you will have to experiment to find out when and how to use them.

Related Patterns

If your program visualisation tool scales enough to accommodate the system your facing, then you can start to VISUALIZE THE STRUCTURE right away. However, since program visualisations rarely scale well, it is preferable to first filter out which parts of the source code are relevant for further analysis. Therefore, consider to INSPECT THE LARGEST (p. 131) or to EXPLOIT THE CHANGES (p. 135) before you VISUALIZE THE STRUCTURE.

Resulting Context. By applying this pattern, you will have obtained an overview of the structure of a selected part of a software system, including potential design anomalies. Some other patterns may help you to further analyse these constructs. For instance, if you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour and if you CHECK METHOD INVOCATIONS (p. 140) you can find out how a class is related to other classes. If you have identified design anomalies, you should consider to refactor them (see [FOWL 99]). Some typical design anomalies including the way to refactor them can be found in the part on Reengineering — p.163.

CHECK METHOD INVOCATIONS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Find out how a class is related to other classes by checking the invocations of key methods in the interface of that class. Two examples of key methods that are easy to recognise are constructors and overridden methods.

> ***Example.*** *You have identified a number of classes that represent part of the domain model. You want to learn about the aggregation relationships between these classes and therefore, you will inspect for all constructor methods which methods are invoking them.*

> ***Example.*** *You have identified a part of a class hierarchy where the designers relied on template methods to customise the design. To learn how the subclasses interact with their superclasses, you will retrieve all methods overriding another one, and inspect who is invoking these methods.*

Context

You are in a later stage of reverse engineering a software system implemented in an object-oriented language. You have an overall understanding of the system's functionality and based on that understanding, you have selected a part of the class hierarchy for further inspection. You have a code browser at your disposal that allows you to jump from a method invocation to the places where the corresponding method is defined.[3]

Problem

You want to find out how a class is related to the other classes in the system.

Solution

Select key methods in the interface of the class and inspect who is invoking these methods.

Variations. The pattern has two variants depending on the selected methods in the public interface.

- *The constructor method variant* suggests you to look at invocations of constructor methods to reveal aggregation relationships between classes.
- *The template method variant* recommends you to select methods that are overridden in a subclass plus the methods invoking them to infer template methods [GAMM 95].

[3] Note that your code browser should take polymorphism into account. Polymorphism implies that one invocation has several candidates for being the defining method. Because the actual target can only be resolved at run-time, your browser must show all candidates.

Hints. If you consider applying the above variants, following suggestions may help you getting the best out of your efforts.

- For *the constructor method variant*, you must trace the chain of invocations until the result of the constructor is stored into an attribute. The class defining this attribute is the aggregation. Also, look out for invocations of constructor methods where the invoking object is passing itself as an argument and where this argument is stored into an attribute of the constructor class. In that case, the constructor class is the aggregation.
- *The template method variant*, explicitly states that you should look a methods that are overridden and not methods that are declared abstractly. The reason is that not all template methods distinguish the hook method via an abstract method, but that often a concrete method is used to specify the default behaviour. By looking for overridden methods, you are certain that you will cover the latter case as well.

Rationale

If the object-oriented paradigm is applied well, state should be encapsulated behind the interface of a class (see [MEYE 97] and [BECK 97] among others). Therefore, to understand how a class is related to other classes, method invocations are more reliable than attribute declarations. Yet, because the amount of method invocations is large you must choose which invocations to analyse. This pattern helps you in this choice by suggesting two specific kinds of methods that are easy to identify and result in well-known class relationships.

Known Uses

In [DEME 98a] we report on a case study where we applied the template method variant.

Related Patterns

Checking method invocations of classes is quite tedious, thus it is best to start with a small amount of classes. Therefore, consider to INSPECT THE LARGEST (p. 131) or EXPLOIT THE CHANGES (p. 135) or VISUALIZE THE STRUCTURE (p. 138) to limit the amount of classes to inspect.

Resulting Context. By applying this pattern, you will know how a class is related to the other classes in the system. If you STEP THROUGH THE EXECUTION (p. 142) you will get a better perception of the run-time behaviour of these relationships.

STEP THROUGH THE EXECUTION

Intent

Obtain a detailed understanding of the run-time behaviour of a piece of code by stepping through its execution.

> ***Example.*** *You have a piece of code that implements a graph layout algorithm and you must understand it in order to rewrite it. You will feed a graph into the program and use the debugger to follow how the algorithm behaves.*

Context

You are in a later stage of reverse engineering a software system. You have an overall understanding of the system's functionality and based on that understanding, you have selected part of the software system for further inspection. You have a debugger at your disposal that allows you to inspect data structures and to interactively follow the step by step execution of a piece of code. You know a set of representative input data to feed into that piece of code to launch a normal operation sequence.

Problem

You want to obtain a detailed understanding of the run-time behaviour of a piece of code.

Solution

Feed representative input data in the piece of code to launch a normal operation sequence. Use the debugger to follow the step by step execution and to inspect the internal state of the piece of code.

Hints.

- Test programs usually provide representative input data in their initialisation code.
- Usage scenarios, like the ones resulting from INTERVIEW DURING DEMO (p. 117), may give clues on what is a normal operation sequence.

Forces Resolved

Limited Resources Once you know what you really want to understand, this pattern works well in a limited resource context. However, stepping through the code can be highly inefficient whitout a clear focus.

Tools and Techniques The success of this pattern relies on the ability to use a good interactive debugger.

Reliable Info By following the step by step execution of a program, you get a very reliable view of a piece of code. However, beware that the input data is indeed representative for a normal operation sequence.

Abstraction The abstraction level is quite low, unless you can tie the step by step execution to a typical usage scenario.

Sceptic Colleagues Neutral.

Rationale

In [MEYE 97], object-oriented programming is defined as "designing a system around the functionality it offers rather then the data structures it operates upon". Hence, understanding the run-time behaviour is crucial to understand an object-oriented program. And the best way to get a view on the run-time behaviour is to see the events as they actually occur in a real execution, a view which is provided by interactive debugging tools.

Known Uses

In [ROCH 93] you can find some interesting debugging techniques applicable in the context of Smalltalk. Many of them will generalise to other programming environments as well.

Related Patterns

Stepping through program executions is quite tedious, thus it is best to focus on a small piece of code. Consider to INSPECT THE LARGEST (p. 131) or to EXPLOIT THE CHANGES (p. 135) or to VISUALIZE THE STRUCTURE (p. 138) to obtain such a focus. Also, you need some typical usage scenarios which may be provided by INTERVIEW DURING DEMO (p. 117).

Resulting Context. By applying this pattern, you will have a detailed understanding of the run-time behaviour of a piece of code. This may be necessary to apply patterns in PREPARE REENGINEERING (p. 145).

Chapter 7

Cluster: Prepare Reengineering

The reverse engineering patterns in this cluster are only applicable when your reverse engineering activities are part of a larger reengineering project. That is, your goal is not only understanding what's inside the source code of a software system, but also rewriting parts of it. Therefore, the patterns in this cluster will take advantage of the fact that you will change the source code anyway.

	Limited Resources	Tools and Techniques	Reliable Info	Abstraction	Sceptic Colleagues
WRITE THE TESTS	- -	-	++	++	0
REFACTOR TO UNDERSTAND	- -	-	0	+	0
BUILD A PROTOTYPE	- -	-	+	- -	++
FOCUS BY WRAPPING	- -	0	0	0	0

Table 7.1: How each pattern of PREPARE REENGINEERING resolves the forces. Very good: ++, Good: +, Neutral: 0, Rather Bad: -, Very bad: - -

WRITE THE TESTS

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Record your knowledge about how a component reacts to a given input in a number of black box tests, this way preparing future changes to the system.

> ***Example.*** *You are asked to extend a parser for a command language so that it is able to parse two additional commands. Before actually changing the of parser, you will write a number of test programs that check whether the parser accepts all valid command sequences and rejects some typical erronous ones.*

Context

You are at the final stages of reverse engineering a software system, just before you will start to reengineer a part of that system. You have sufficient knowledge about that part to predict its output for a given input.

Problem

Before starting to reengineer the system, you want to make sure that all what used to work keeps on working.

Solution

Write a number of black box tests that record your knowledge about the input/output behaviour.

REFACTOR TO UNDERSTAND

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Obtain better readable —thus more understandable— and better organised —thus more extensible— code via renaming and refactoring.

> ***Example.*** *You are asked to extend a parser for a command language so that it is able to parse two additional commands. Before actually extending the parser, you will improve the readability of the source code. Among others, you will rename key methods and classes to reflect your understanding of a parser and you will split long and complex methods into smaller ones. As an example of the former, you will rename the class* StreamIntf *into* Scanner *and the method* rdnxt *into* nextToken*. An example of the latter would be to split the* nextToken *method, so that it becomes a large case statement, where each branch immediately invokes another method.)*

Context

You are at the final stages of reverse engineering a software system, just before you will start to add new functionality to that system. You have a good programming environment that allows you to rename things easily and that operates on top of a version control system.

Problem

The shape of the code is such that it is difficult to read —hence to understand— and difficult to add the new functionality.

Solution

Reorganise the code so that its structure reflects better what the system is supposed to do.

BUILD A PROTOTYPE

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Extract the design of a critical but cryptic component via the construction of a prototype which later may provide the basis for a replacement.

> ***Example.*** *You have a piece of code that implements a graph layout algorithm. You have an idea on how the algorithm works, but the code is too cryptic to map your knowledge of the algorithm onto the code. You will write a prototype that implements your understanding of the algorithm and map pieces of your code onto the existing code.*

Context

Problem

Solution

FOCUS BY WRAPPING

Author(s): Serge Demeyer, Stéphane Ducasse and Sander Tichelaar

Intent

Wrap the parts you consider unnecessary for the future reengineering in a black box component.

> ***Example.*** *You have to migrate a graph manipulation program from a Unix to Macintosh user-interface platform. The original program is well designed and has separated out most of the platform specific operations into a separate layer. You will clean up this layering by moving* ***all*** *platform specific behaviour into a separate layer, this way wrapping the obsolete part into a separate component.*

Context

Problem

Solution

Chapter 8

Cluster: misc

CONFER WITH COLLEAGUES

Intent

Share the information obtained during each reverse engineering activity to boost the collective understanding about the software system.

> ***Example.*** *Your team has to reverse engineer a workflow system containing lots of complex rules on how tasks get transferred. Each team member investigates a part of the system and as such the knowledge about the workflow rules is distributed over the team. To increase the overall understanding, you will devote 15 minutes of the weekly team meeting to discuss reverse engineering results made during the last week.*

Context

You are a member of a software development team and part of the job assigned to your team is the reverse engineering of a software system. Different members of the team perform different reverse engineering activities and consequently the knowledge about the system is scattered throughout the team.

Problem

How do you ensure that every team member contributes to the overall understanding of the software system.

Solution

Use whatever means at your disposal (meetings, e-mail, intra-nets, ...) to ensure that whenever any team member finishes a reverse engineering step, the obtained information is shared with the rest of the team.

Hints.

- To avoid information overload, choose the communication channels in such a way that sharing the information fits well with the culture within your team. For instance, do not organise a special team meeting devoted to reverse engineering results; rather use an existing meeting as a vehicle for applying this pattern.

Rationale

Reverse engineering is sometimes compared with solving a puzzle [Will96b]. If team members keep some pieces of the puzzle for themselves it will never be possible to finish the puzzle. Consequently, it is imperative that a reverse engineering team is organised in such a way that information may be shared among the various team members.

Chapter 9

Pattern Overview

The followin tables summarize the patterns for reference purposes.

The first series of tables lists the patterns together with their problem and their solution, this way aiding reverse engineers to identify which patterns may be applicable to their problem.

The second series of tables show how all the patterns work together to tackle an overall reverse engineering project. For each pattern, the tables list the context and prerequisites plus the pattern results and how these results may serve as input for other patterns.

Pattern	Problem	Solution
	FIRST CONTACT	
READ ALL THE CODE IN ONE HOUR (p. 111)	You need an initial assessment of the internal state a software system to plan further reverse engineering efforts.	Grant yourself a reasonably short amount of study time to walk through the source code. Afterwards produce a report including a list of (i) the important entities; (ii) the coding idioms applied ; (iii) the suspicious coding styles discovered
SKIM THE DOCUMENTATION (p. 114)	You need an initial idea of the functionality provided by the software system in order to plan further reverse engineering efforts.	Grant yourself a reasonably short amount of study time to scan through the documentation. Afterwards produce a report including a list of (i) the important requirements; (ii) the important features (iii); the important constraints; (iv) references to relevant design information.
INTERVIEW DURING DEMO (p. 117)	You need an idea of the typical usage scenario's plus the main features of a software system in order to plan further reverse engineering efforts.	Observe the system in operation by seeing a demo and interviewing the person who is demonstrating. Afterwards produce a report including a list of (i) some typical usage scenarios or use cases; (ii) the main features offered by the system and whether they are appreciated or not; (iii) the system components and their responsibilities; (iv) bizarre anecdotes that reveal the folklore around using the system.
	EXTRACT ARCHITECTURE	
Pattern	Problem	Solution
GUESS OBJECTS (p. 123)	You must gain an overall understanding of the internal structure of a software system and report this knowledge to your collegues so that they will use it as a kind of roadmap for later activities.	Based on your experience, and the little you already understand from the system, devise a model that serves as your initial hypotheses of what to expect in the source code. Check these hypotheses against the source code, refine the initial model and recheck the hypotheses. Afterwards, produce a boxes- and arrows diagram describing your findings.
CHECK THE DATABASE (p. 126)	You want to derive a data model for the persistent data in a software system in order to guide further reverse engineering efforts.	Check the database schema to reconstruct at least the persistent part of the data model. Use your knowledge of how constructs in the implementation language are mapped onto database constructs to reverse engineer the real data model. Use the samples of data inside the database to refine the data-model.

FOCUS ON HOT AREAS		
Pattern	Problem	Solution
INSPECT THE LARGEST (p. 131)	You must identify those places in the source code that correspond with important chunks of functionality.	Use a metrics tool to collect a limited set of measurements for all the constructs in the system. Sort the resulting list according to these measurements. Browse the source code for the largest among those constructs in order to understand how these constructs work together with other related constructs. Produce a list of all the constructs that appear important, including a description of how they should be used (i.e. external interface).
EXPLOIT THE CHANGES (p. 135)	You must identify those parts in the design that played a key role during the system's evolution.	Use whatever means at your disposal to compile a list of targets of important/frequent changes. For each target, put yourself in the role of the original developer and ask yourself what the change is about and why it was necessary. With this insight, produce a list of crucial system parts, including a description of the design issues that makes them important.
VISUALIZE THE STRUCTURE (p. 138)	You want to obtain insight in the structure of a selected part of a software system, including knowledge about potential design anomalies.	Instruct the program visualisation tool to show you a series of graphical layouts of the program structure. Based on these graphical layouts, formulate yourself some assumptions and use the code browser to check whether your assumptions are correct. Afterwards, produce a list of correct assumptions, classifying the items in one of two categories: (i) helps program understanding, or (ii) potential design anomaly.
CHECK METHOD INVOCATIONS (p. 140)	You want to find out how a class is related to the other classes in the system.	Select key methods in the interface of the class and inspect who is invoking these methods. Two examples of key methods that are easy to recognise are constructors and overridden methods.
STEP THROUGH THE EXECUTION (p. 142)	You want to obtain a detailed understanding of the run-time behaviour of a piece of code.	Feed representative input data in the piece of code to launch a normal operation sequence. Use the debugger to follow the step by step execution and to inspect the internal state of the piece of code.

FIRST CONTACT

Context: You are starting a reverse engineering project of a large and unfamiliar software system.

Pattern	*Prerequisites*	*Result*	*What next ?*
READ ALL THE CODE IN ONE HOUR (p. 111)	• source code • expertise with the implementation language	• the important entities (i.e., classes, packages, ...) • the coding idioms applied • the suspicious coding styles discovered	• SKIM THE DOCUMENTATION (p. 114) and INTERVIEW DURING DEMO (p. 117) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to refine the list of important entities
SKIM THE DOCUMENTATION (p. 114)	• documentation • you are able to interpret the diagrams and formal specifications contained within	• important requirements • important features • important constraints • references to relevant design information. + an assessment of the reliability and usefulness for each of the above.	• READ ALL THE CODE IN ONE HOUR (p. 111) and INTERVIEW DURING DEMO (p. 117) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to map the information on an overall system blueprint.
INTERVIEW DURING DEMO (p. 117)	• running system • somebody who can demonstrate how to use the system	• typical usage scenarios or use cases • the main features offered by the system and whether they are appreciated or not • the system components and their responsibilities • bizarre anecdotes that reveal the folklore around using the system	• READ ALL THE CODE IN ONE HOUR (p. 111) and SKIM THE DOCUMENTATION (p. 114) to get alternative views • CONFER WITH COLLEAGUES (p. 152) to report findings • GUESS OBJECTS (p. 123) and CHECK THE DATABASE (p. 126) to map the information on an overall system blueprint.

EXTRACT ARCHITECTURE

Context: You are in the early stages of reverse engineering a software system. You have an initial understanding of its functionality and you are somewhat familiar with the main structure of its source code. (This initial understanding might have been obtained by the patterns in FIRST CONTACT (p. 109)).

Pattern	*Prerequisites*	*Result*	*What next ?*
GUESS OBJECTS (p. 123) (variants: guess patterns, guess object responsabilities, guess object roles, guess process architecture)	• knowledge of important aspects of a software system • on-line access to the source code plus the necessary tools to manipulate it • reasonable expertise with the implementation language being used	• a series of blueprints, each one containing a perspective on the whole system	• CHECK THE DATABASE (p. 126) if you are interested in the data model • all patterns in FOCUS ON HOT AREAS (p. 129) if you want to refine the blueprints
CHECK THE DATABASE (p. 126)	• software system employs some form of a database • you have access to the database, including the proper tools to inspect its schema and samples of the data • knowledge of how data-structures from your implementation language are mapped onto the data-structures of the underlying database	• a data model of the persistent part of your system	• GUESS OBJECTS (p. 123) if you need to obtain other overall bluebrints of the system • all patterns in FOCUS ON HOT AREAS (p. 129) if you want to refine the datamodel

FOCUS ON HOT AREAS

Context: You are in a later stage of reverse engineering a software system. You have an overall understanding of its functionality and you are fairly familiar with the main structure of its source code.

Pattern	*Prerequisites*	*Result*	*What next ?*
INSPECT THE LARGEST (p. 131)	• a code browser • The metrics tool is configured with a number of size metrics	• a list of constructs representing important functionality	• VISUALIZE THE STRUCTURE (p. 138) to obtain other perspectives on those constructs. • STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother • refactoring if you want to split some of these larger contructs into smaller ones
EXPLOIT THE CHANGES (p. 135) (variants:configuration database, change metrics)	• several releases of the source code • a configuration management system and/or a metrics tool	• a list of design parts that played a key role during the system's evolution	• VISUALIZE THE STRUCTURE (p. 138) to obtain other perspectives on those constructs. • STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother
VISUALIZE THE STRUCTURE (p. 138)	• a part of the software system • a program visualisation tool • a code browser	• insight in the selected part • list of potential design anomalies	• STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour. • (in the case of object-oriented source code) CHECK METHOD INVOCATIONS (p. 140) to find out how classes are related to eachother • refactoring if you want to split some of these larger contructs into smaller ones

CHECK METHOD INVOCATIONS (p. 140) (variants: contructor methods, overridden methods)	• a part of the software system • a program visualisation tool • a code browser that allows you to jump from a method invocation to the places where the corresponding method is defined	• a list of classes and the relationships between them	• STEP THROUGH THE EXECUTION (p. 142) to get a better perception of the run-time behaviour.
STEP THROUGH THE EXECUTION (p. 142)	• a part of the software system • an interactive debugger • a set of representative input data	• insight into the run-time behaviour of a piece of code	• PREPARE REENGINEERING (p. 145) if you need to reengineer that piece of code

Part III

Reengineering

Chapter 10

Reengineering Patterns

A *reengineering pattern* describes how to go from an existing *legacy* solution to a *refactored* solution that better suits the current requirements. In this chapter we explain why we choose the pattern form to communicate reengineering expertise and present the reengineering pattern form. We stress the differences between the Design Patterns and the Reengineering Patterns, and also between the Reengineering Patterns and the AntiPatterns. After that we present the differences between the reengineering patterns themselves. It should be noted that the reengineering patterns are not linked together in a pattern language due to a lack of time.

10.1 Reengineering Patterns: a Need

Reengineering projects, despite their diversity, often encounter some typical problems again and again. These can be problems at different levels and due to different practices [FOOT 97]. But it is unlikely that one methodology or process will be appropriate for all projects and organisations [STEV 98], just as not one tool or technique can be expected to solve all the technical problems encountered in a reengineering project. To allow reengineering projects to benefit from the experience gained in previous efforts, an appropriate form is required for transferring expertise. This form should be small enough to be easily consulted and navigated, and stable enough as to be useful for many reengineering projects.

In the object-oriented software engineering community *Design Patterns* [GAMM 95] have been adopted as an effective way of communicating expertise about software design. A design pattern describes a solution for a recurring design problem in a form which facilitates the reuse of a proven design solution. In addition to the technical description of the solution, an important element of a design pattern is its discussion of the advantages and disadvantages of applying the pattern.

We propose the use of the pattern form as a means of communicating expertise in the area of reengineering. *Reengineering patterns* codify and record knowledge about modifying legacy software: they help to diagnose problems and identify weaknesses hindering further development of the system as well as aiding the search for solutions that better satisfy the new requirements. We see reengineering patterns as stable units of expertise which can be consulted in any reengineering effort: they describe a process without proposing a complete methodology, and they suggest appropriate tools without 'selling' a specific one. A more thorough discussion of the advantages of the pattern form as a vehicle for reengineering expertise can be found in [STEV 98], which discusses patterns closely related to ours.

All the reengineering patterns presented hereafter address problematic legacy solutions typically found in object-oriented code, and describe how to move from the *legacy* solution to a new *refactored* solution. The patterns presented are all of a technical nature, dealing directly with source code problems. There exist however higher-level reengineering patterns which describe overall strategies for dealing with legacy

systems. See for instance the *Systems Reengineering Patterns* [STEV 98] which address broader methodological issues. The 'Deprecation' pattern [STEV 98], for example, describes how to iteratively change interfaces of a system in a friendly way for the client of the system under change.

10.2 Reengineering Patterns and Related Work

The reengineering patterns presented here and the *Systems Reengineering Patterns* of [STEV 98] are closely. The principle differences is that here the patterns are source-level rather high-level and they are focused on bject-oriented legacy systems[1]. Note that our patterns cannot be used to evaluate whether or not an application should be reengineered in the first place; this difficult task has been tackled by [STS 97] and [RANS 98]. In [BROD 95] a methodology is proposed to help in the migration of legacy systems (principally legacy database systems) to new platforms.

Reengineering patterns differ from *Design Patterns* [GAMM 95] in their emphasis on the *process* of moving from an existing *legacy* solution that is no longer appropriate to a new *refactored* solution. The mark of a good reengineering pattern is (a) the clarity with which it exposes the advantages, the cost and the consequences of the target solution with respect to the existing solution, and not how elegant the target solution is, and (b) the description of the change process: how to get from the legacy version of the system to the refactored version.

We also contrast reengineering patterns with *AntiPatterns* [BROW 98]. Antipatterns, as exposed by Brown et al., are presented as "bad" solutions to design and management issues in software projects. Many of the problems discussed are managerial concerns that are outside the direct control of developers. Moreover, the emphasis in antipatterns is on prevention: how to avoid making the mistakes which lead to the "bad" solution. Consequently, antipatterns may be of interest when starting a project or during development but are no longer helpful when we are confronted with a legacy system. In reengineering, though, we prefer to withhold the judgement inherent in the notion of "bad solution" and use the term "legacy solution" or "legacy pattern" for a solution which at the time, and under the constraints given, seemed appropriate. In reengineering it is too late for prevention, and reengineering patterns therefore concentrate on the cure: how to detect problems and move to more appropriate solutions.

Finally, our reengineering patterns are different from code *refactorings* [JOHN 93, JOHN 93]. A reengineering pattern describes a process which starts with the detection of the symptoms and ends with the refactoring of the code to arrive at the new solution. A refactoring is only the last stage of this process, and addresses only the technical issue of automatically or semi-automatically modifying the code to implement the new solution. Reengineering patterns also include other elements which are not part of refactorings: they emphasise the context of the symptoms by taking into account the constraints being faced and include a discussion of the impact of the changes introduced by the refactored solution.

A reengineering pattern may describe a solution that would not be ideal if one is designing a system from scratch, but is a good solution under the current constraints of the legacy system. For example, if the constraint is that changes must be kept local some solutions are clearly not applicable even if they seem at first hand to be the best solutions.

10.3 Form of a reengineering pattern

The primary goal of a reengineering pattern is to help developers in solving reengineering problems. The idea being that a developer must diagnose a problem, identify the available options and choose a particular course of action. Furthermore, the relevant weaknesses must be identified, where relevant is defined in terms of the desired flexibility or some other quality, and the system transformed so that it possesses the desired quality.

[1] We do not address problems of reengineering procedural applications to object-oriented ones.

The pattern form has been defined principally for *reengineering* patterns: that is, patterns which describe a transformation of a existing design to a more appropriate, target design. In general the pattern form has been defined with the following requirements in mind, although some patterns may - due to specific needs - add or omit sections described in the presented pattern format:

Focus on Reengineering Process. A reengineering pattern is different from a design pattern. It should go beyond discussing good and bad designs. A reengineering pattern should also discuss the reengineering *process*! For example if we know a design to contain problems according to present requirements, then how can these problems be discovered; or, what are the pitfalls in transforming a system? Both are issues that have to do with the reengineering process itself. A typical reengineering pattern will describe a process that transforms a system with a design that is no longer adequate into a system with an improved design. The reengineering pattern should clearly identify these two states of the system and their relationship.

Easy Navigation. The idea is that a handbook user (i.e. reengineer) should be able to determine if a pattern is applicable within the *first* page of description.

Separate out Tool and Language Dependent Issues. To make the patterns as generally applicable as possible, tool and language dependent issues should be separated out as much as possible. The main part of the patterns describes stable reengineering knowledge, whereas tools are more subject to evolution and in some cases language dependent issues can be interesting but not of influence on the core idea of the pattern.

Standard Terminology and Notation. A language neutral terminology and notation is mandatory if the patterns are to be kept as language independent as is reasonably possible. The rule for terminology is: as far as it is defined the UML terminology [SOFT 97] is used for object oriented concepts and if a term is not defined by UML then the terminology of the FAMOOS Information Exchange model (see section 14.1) is used. All other terms are to be defined in a glossary that is part of this handbook. For the homogeneity of the patterns, a strong requirement is that all the drawings should be done using UML notations.

The specific aspects of reengineering patterns lead us to the definition of an adapted form for the reengineering patterns. This form is structured as follows:

Pattern Name. The name is based on the reengineering operation that is performed as this is the most natural way of discussing the pattern in the context of reengineering. It must form the basis for a terminology for reengineers to talk about reengineering a system. As a temporary solution, patterns which miss a good word will be named by a short sentence with a verb that emphasises the kind of reengineering transformation.

Intent. A description of the process, together with the result and why it is desirable.

Applicability. When is the pattern applicable? When is it not applicable? This section includes a list of symptoms, a list of reengineering goals and a list of related patterns. Symptoms are those experienced when reusing, maintaining or changing the system. For example, correlations between editing different parts of a system for making a certain change can indicate the need for a particular kind of reengineering. Reengineering goals present the qualities improved through the application of this pattern.

Motivation. This section presents an example: it must acquaint the reader with a concrete example so they can better understand the more abstract presentation of the problem which follows in the structure and process sections. The example must clearly describe the structure of the legacy system, the structure of the reengineered system, and the relation between the two. The state of the system before and after the application of the pattern are described.

Structure. This section describes the structure of the system before and after reengineering. Each structure section is similar to the structure section in the Gang of Four pattern book. The participants and their collaborations are identified. Consequences discuss the advantages and disadvantages of the target structure in comparison to the initial structure.

Process. The process section is subdivided into three sections: the detection, the recipe and the difficulties. The detection section describes methods and tools to detect that the code is indeed suffering from the suspected problem and that the process given below can help to alleviate this problem. The recipe states how to perform the reengineering operation and the possible variants. The optional difficulties subsection discusses situations where the reengineering operation is not feasible or is compromised by other problems.

Discussion. In this section, forces of the legacy solution are discussed first. Indeed, often legacy solutions fulfilled the requirements at the time they were implemented. When requirements change however, the solution must evolve to accommodate the new requirements. The forces of the refactored solution are stressed in terms of cost and benefit tradeoffs of applying the pattern. What is the cost of detecting this problem? What is the magnitude of the problem? What is the benefit gained by applying the pattern? This discussion should aid an engineer in deciding (once he knows the pattern is applicable to the code) whether or not it is, in this specific case, worth applying the pattern. Moreover, relationships with Design Patterns or AntiPatterns should be documented.

The sections above form the core of the pattern. The sections described below deal with more concrete issues and are essential to the reengineering handbook, where engineers need information about language specific issues and existing tool support and know applications.

Language Specific Issues. This section lists what must be specifically resolved for each language. What makes it more difficult? More easy?

Tool Support. Lists and describes tools to support the detection of symptoms, detection of participants and collaborations, and to aid in the transformation of the system.

Known Uses. Gives known cases where the pattern has been applied, successfully or not. In our context, references are made to our industrial case studies.

10.4 Pattern Navigation

The patterns that we have collected for this handbook solve a quite random set of reengineering problems. To provide the reader with some guidance , we present ways to select the patterns that might be of interest in solving a specific problem of yours. The usefulness of the navigational support is inherently limited due to the rather small set of available patterns, but it provides an overview of what is already covered by the patterns.

We currently provide navigation based based on forces that typically play a role in reengineering (see section 10.4.1). For each type of navigation, a table is shown that gives a broad overview of which pattern covers what. Following the table, the patterns are listed – according to the same structure as the table – with a short explanation of their appearance in the table. This allows for a quick scan of the available patterns.

10.4.1 Forces

In this section we categorise the patterns according to how they affect the forces that are at work in a reengineering project. In table 10.1 we show the reengineering patterns and their influence on the different forces. The patterns are shown vertically and the forces horizontally. A ‘+’ means that the pattern increases

the effect of, or has a high impact on, the particular force. A '-' means that the pattern reduces the impact of, or has a low impact on, the force. No sign means that the pattern either has no or an unpredictable influence on the force. To give an example, the Code Duplication Detection pattern requires only minor effort when applying it, scales up well and does not require much parsing.

	Forces						
	Flexibility	Understandability	Reusability	Effort	Scalability	Parsing effort	Global Impact
Type Check Elimination in Clients	+	+	+				-
Type Check Elimination within a Provider Hierarchy	+	+					
Detection of Duplicated Code				-	+	-	
Repairing a Broken Architecture	+	+					
Transforming Inheritance into Composition	+		+				
Distribute Responsibilities	+	+	+				

Table 10.1: How the individual patterns affect the forces of a reengineering effort.

Flexibility

TYPE CHECK ELIMINATION IN CLIENTS. By reducing the coupling between the clients and the provider class hierarchy by refactoring the interface of the provider classes and the client code that depends on these interfaces thus making the clients much more robust. This greatly facilitates extending the functionality of the provider hierarchy without breaking the client code.

TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY. Transforms a single provider class being used to implement what are conceptually a set of related types into a hierarchy of classes. Decision structures, such as case statements or if-then-elses, over type information are replaced by polymorphism. This results in increased modularity and facilitates the extension of functionality through the addition of new subclasses.

REPAIRING A BROKEN ARCHITECTURE. Detects and removes dependencies between packages of a system that aren't allowed according to the designated system architecture. These architecture breaking dependencies may prohibit the exploitation of the architecture's advantages and cause unexpected effects at maintenance work.

TRANSFORMING INHERITANCE INTO COMPOSITION. This pattern describes, how to transform an inheritance relationship into a component relationship. This increases flexibility, because a component relationship can be changed dynamically whereas an inheritance relationship can only be changed statically.

DISTRIBUTE RESPONSIBILITIES. Distributes the responsibilities equally among the classes of an object-oriented system to prevent large, hardly maintainable and reusable classes.

Understandability

TYPE CHECK ELIMINATION IN CLIENTS. The reduced coupling between client and provider classes as well as the refactored interface of the provider classes present a more modular view of what is essentially the same functionality thus facilitating understandability.

TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY. Breaking a single complex class into a hiearchy of simpler but more specialised classes facilitates partial understanding rather than requiring complete understanding. This simplifies understanding how the hierarchy can be extended by separating information that is relevant to the entire hierarchy from that which is specific to just a few classes.

REPAIRING A BROKEN ARCHITECTURE. Detects and removes dependencies between packages of a system that aren't allowed according to the designated system architecture. These architecture breaking dependencies may prohibit the exploitation of the architecture's advantages and cause unexpected effects at maintenance work.

DISTRIBUTE RESPONSIBILITIES. Distributes the responsibilities equally among the classes of an object-oriented system to prevent large, hardly maintainable and reusable classes.

Reusability

TYPE CHECK ELIMINATION IN CLIENTS. The refactored interface of the classes from the provider hierarchy more accurately reflect the needs of any client classes. This increases the likelyhood that classes from the provider hierarchy can be reused while simplifying their reuse.

TRANSFORMING INHERITANCE INTO COMPOSITION. This pattern describes, how to transform an inheritance relationship into a component relationship. This increases flexibility, because a component relationship can be changed dynamically whereas an inheritance relationship can only be changed statically.

DISTRIBUTE RESPONSIBILITIES. Distributes the responsibilities equally among the classes of an object-oriented system to prevent large, hardly maintainable and reusable classes.

Relatively minor effort

DETECTION OF DUPLICATED CODE. Collecting duplication data is fully automatised. Filtering the data to find candidates for refactoring automatically is possible only in some special cases. Normally, human intervention and expertise is required to assess the duplication and decide on the possible refactoring operations.

Patterns that easily scale up

DETECTION OF DUPLICATED CODE. Code Duplication is done with tool support. All of the tools are built to scale up well.

Patterns that need a low parsing effort

DETECTION OF DUPLICATED CODE. Depending on the technique used to detect duplication, more or less parsing is required. The DUPLOC tool (see Chapter 17) developed in FAMOOS requires only minimal parsing.

Global impact

TYPE CHECK ELIMINATION IN CLIENTS. The pattern involves refactoring the interface of a class hierarchy in order to better support the clients of these classes. Consequently, most clients will be affected which may potentially require that changes be made throughout the system.

Higher number of Classes

DISTRIBUTE RESPONSIBILITIES. Distributes the responsibilities equally among the classes of an object-oriented system to prevent large classes that are difficult to maintain and reuse.

TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY. Separates a single complex class into a hiearchy of simpler more specialized classes representing a cleaner separation of concerns. The increased number of classes allow a greater precision in expressing dependencies.

Chapter 11

Cluster: Type Check Elimination

TYPE CHECK ELIMINATION IN CLIENTS

Author(s): Stéphane Ducasse, Robb Nebbe and Tamar Richner

Intent

Transform *client* classes that depend on type tests (usually in conjunction with case statements) into *clients* that rely on polymorphism. The process involves factoring out the functionality distributed across the clients and placing it in the provider hierarchy. This results in lower coupling between the *clients* and the *providers* (class hierarchy).

Applicability

Symptoms.

- Large decision structures in the *client* over the type of (or equivalent information about) an instance of the *provider*, either passed as an argument to the client, an instance variable of the client, or a global variable.
- Adding a new subclass of the *provider* superclass requires modifications to *clients* of the *provider* hierarchy because functionality is distributed over these clients.

Reengineering Goals.

- Localise functionality distributed across *clients* in the *provider* hierarchy.
- Improve usability of *provider* hierarchy.
- Lower coupling between *clients* and the *provider* hierarchy.

Related Reengineering Patterns. A closely related reengineering pattern is TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY, where the case statements over types are in the *provider* code as opposed to the *client* code. The essential distinction is if the decision structure is over the type or an attribute functioning as a type of: (a) an instance of *another* class (this pattern) or (b) an instance of the class to which the method belongs (see TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY in section 11).

Motivation

The fact that the clients depend on provider type tests is a well known symptom for a lack of polymorphism. This leads to unnecessary dependencies between the classes and it makes it harder to understand the program because the interfaces are not uniform. Furthermore, adding a new subclass requires all clients be adapted.

Initial Situation. The following code illustrates poor use of object-oriented concepts as shown by Fig. 11.1. The function makeCalls takes a vector of Telephone's (which can be of different types) as a parameter and makes a call for each of the telephones. The case statement switches on an explicit type-flag returned by phoneType(). In each branch of the case, the programmer calls the phoneType specific methods identified by the type-tag to make a call.

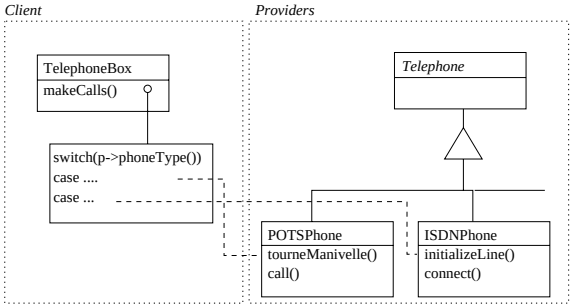


Figure 11.1: Initial relation and structure of clients and providers.

```
void makeCalls(Telephone * phoneArray[])
{
  for (Telephone *p = phoneArray; p; p++) {
    switch(p->phoneType()) {
    case TELEPHONE::POTS: {
      POTSPhone * potsp = (POTSPhone *) p;
      potsp->tourneManivelle();
      potsp->call(); break;}
    case TELEPHONE::ISDN: {
      ISDNPhone * isdnp = (ISDNPhone *) p;
      isdnp->initializeLine();
      isdnp->connect(); break;}
    case TELEPHONE::OPERATORS: {
      OperatorPhone * opp = (OperatorPhone *) p;
      opp->operatormode(on);
      opp->call(); break;}
    case TELEPHONE::OTHERS:
    default:
        error(....);
    } } }
```

Final Situation. After applying the pattern the corresponding ringPhones() will look as follows and the structure as shown by the Fig. 11.2.

```
void makeCalls(Telephones *phoneArray[])
{
  for(Telephone *p = phoneArray; p; p++) p->makeCall();
}
```

Note that the client code, which represents distributed functionality, has been greatly simplified. Furthermore, this functionality has been localised within the Telephone class hierarchy, thus making it more complete and uniform with respect to the clients needs.

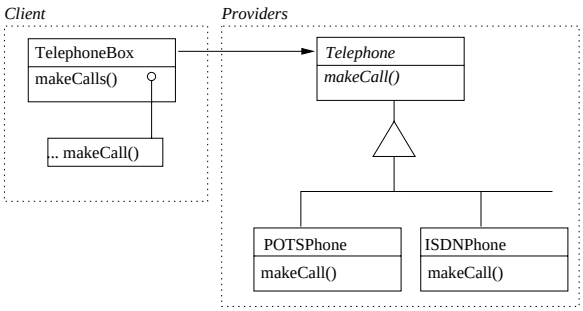


Figure 11.2: Final relation and structure of clients and providers.

Structure

Participants.

- **provider classes** (Telephone and its subclasses)
  - organised into a hierarchy.
- the **clients** (TelephoneBox) of the provider class hierarchy.

Collaborations. The collaborations will change between all clients and the providers as well as the collaboration within the provider hierarchy.

Initially, the clients collaborate directly with the provider superclass and its subclasses by virtue of type tests or a case statement over the types of the subclasses. After reengineering the only direct collaboration between the clients and the providers is through the superclass. Interaction specific to a subclass is handled indirectly through polymorphism.

Within the provider hierarchy the superclass interface must be extended to accurately reflect the needs of the clients. This will involve the addition of new methods and the possible refactorisation of the existing methods in the superclass. Furthermore, the collaborations between the provider superclass and its subclasses may also evolve, i.e. it must be determined whether the new/refactored methods are abstract or concrete.

Consequences. Relying on polymorphism localises the protocol for interacting with the provider classes within the superclass. The collaborations are easier to understand since the interface actually required by the clients is now documented explicitly in the provider superclass. It also simplifies the addition of subclasses since their responsibilities are defined in a single place and not distributed across the clients of the hierarchy.

Process

Detection. The technique described in the pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY to detect case statements is applicable for this pattern. Whereas in the pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY, the switches are located in the same class, hence in one file for a language like C++, in this pattern the case statements occur in several classes which can be spread over different files.

Recipe. The process consists of two major steps. The first is to encapsulate all the responsibilities that are specific to the provider classes within the provider hierarchy. The second is to make sure that these responsibilities are correctly distributed within the hierarchy.

1. Determine the set of clients to which the pattern will be applied.

2. Define a new abstract method in the provider superclass and concrete methods implementing this method in each of the subclasses based on the source code contained within each branch of the case statement.

3. Refactor the interface of the provider superclass to accurately reflect the protocol used by the clients. This involves not only adding and possibly changing the methods included but determining how they work together with the subclasses to provide the required behaviour. This includes determining whether methods are abstract or concrete in the provider superclass.

4. For each client, rewrite the method containing the case statement so that it uses only the interface of the provider superclass.

Difficulties.

1. The set of clients may all employ the same protocol; in this case the pattern needs to be applied only once. However, if the clients use substantially different protocols then they can be divided into different kinds and the pattern must be applied once for each kind of client.

2. If the case statement does not cover all the subclasses of the provider superclass a new abstract class may need to be added and the client rewritten to depend on this new class. For example, if it is an error to invoke the client method with some subclasses as opposed to just doing nothing then the type system should be used to exclude such cases. This reduces the provider hierarchy to the one starting at the new abstract class.

3. Refactoring the interface will affect all clients of the provider classes and must not be undertaken without examining the full consequences of such an action.

4. Nested case statements indicate that multiple patterns must be applied. This pattern may need to be applied recursively in which case it is easiest to apply the pattern to the outermost case statement first. The provider classes then become the client classes for the next application of the pattern. Another possibility is when the inner case statement is also within the provider class but some of the state of the provider classes should be factored out into a separate hierarchy.

Discussion

During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [BECK 94], [ALPE 98]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [GAMM 95] , [ALPE 98]. Moreover, the Strategy pattern [GAMM 95], [ALPE 98] is also based on the elimination of case statement over object state.

Language Specific Issues.

C++ In C++ virtual methods can only be used for classes that are related by an inheritance relationship. The polymorphic method has to be declared in the superclass with the keyword virtual to indicate that calls to this methods are dispatched at runtime. These methods must be redefined in the subclasses.

Since C++ does not offer runtime type information, type information is encoded mostly using some enum type. A data member of a class having such an enum type and a method to retrieve these tags are usually a hint that polymorphism could be used (although there are cases in which polymorphic mechanism cannot substitute the manual type discrimination).

ADA Detecting type tests falls into two cases. If the hierarchy is implemented as a single discriminated record then you will find case statements over the discriminant. If the hierarchy is implemented with tagged types then you cannot write a case statement over the types (they are not discrete); instead an if-then-else structure will be used.

If a discriminated record has been used to implement the hierarchy it must first be transformed by applying the TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY pattern.

SMALLTALK In SMALLTALK the detection of the case statements over types is hard because few type manipulations are provided. Basically, methods isMemberOf: and isKindOf: are available. anObject isMemberOf: aClass returns true if anObject is an instance of the class aClass, anObject isKindOf: aClass returns true if aClass is the class or a superclass of anObject. Detecting these method calls is not sufficient, however, since class membership can also be tested with self class = anotherClass, or with property tests throughout the hierarchy using methods like isSymbol, isString, isSequenceable, isInteger.

Tools

Glimpse and agrep can be found at ftp://ftp.cs.arizona.edu/glimpse.

Known Uses

In the FAMOOS mail sorting case study, we identified 28 matches (a match is not equivalent to a file because a same file may contain several switches) for the expression agrep 'switch;type', 185 matches for the sole expression agrep 'switch'. In the same time agrep 'if' gave us 10976 matches whereas using the perl script shown above we reduce the matches to 497.

In this case study, we identify three obvious lacks of polymorphism but they were not corresponding with the presented pattern but its companion pattern TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY. We found also cases that implement state object [GAMM 95].

This pattern has been also applied in one of the FAMOOS case studies written in Ada. This considerably decreased the size of the application and improved the flexibility of the software. In one of the FAMOOS C++ case studies, manual type check also occurs implemented statically via # ifdefs.

TYPE CHECK ELIMINATION WITHIN A PROVIDER HIERARCHY

Author(s): Stéphane Ducasse, Robb Nebbe and Tamar Richner

Intent

Transform a single *provider* class being used to implement what are conceptually a set of related types into a hierarchy of classes. Decision structures, such as case statements or if-then-elses, over type information are replaced by polymorphism. This results in increased modularity and facilitates the extension of functionality through the addition of new subclasses.

Applicability

Symptoms.

- Methods contain large decision structures over an instance variable of the *provider* class to which they belong.
- Extending the functionality of the *provider* class requires modifying many methods.
- Many *clients* depend on a single *provider* class.

Reengineering Goals.

- Improve modularity.
- Simplify extension of *provider* functionality.

Related Reengineering Patterns. A closely related pattern is TYPE CHECK ELIMINATION IN CLIENTS where the case statements over types are in the client code as opposed to the provider code. The essential distinction is if the decision structure is over an instance variable of the class (this pattern) or another class (see TYPE CHECK ELIMINATION IN CLIENTS in section 11).

Motivation

Case statements are sometimes used to simulate polymorphic dispatch. This often seems to be the result of the absence of polymorphism in an earlier version of the language (Ada'83 $\rightarrow$ Ada'95 or C $\rightarrow$ C++). Another possibility is that programmer don't fully master the use of polymorphism and as a result do not always recognise when it is applicable. In any language that supports polymorphism it is preferable to exploit the language support rather than simulate it.

In the presence of polymorphism the process of dispatching is part of the language. In contrast, with case statements or other large decision structures the simulated dispatch must be hand coded and hand maintained. Accordingly, changing or extending the functionality are more difficult because they often

affect many places in the source code. It also results in long methods with obscured logic that are hard to understand.

Programmers often fall back to the language they are most familiar with – in the Variable State pattern Kent Beck shows an example of such a situation related to Lisp programmers [BECK 97]. Thus, they may continue to implement solutions which do not exploit polymorphism even when polymorphism is available. This could occur especially when programmers extend an existing design by programming around its flaws, rather than reengineering it.

Initial Situation. Our example, taken in a simplified form from one of the case studies, consists of a message class that wraps two different kinds of messages (TEXT and ACTION) that must be serialised to be sent across a network connection as shown in the code and the figure 11.3.

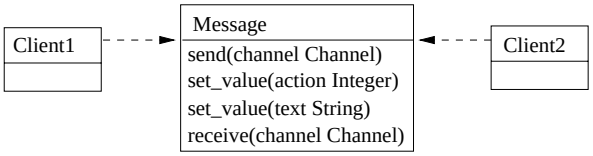


Figure 11.3: Initial relation and structure of clients and providers.

A single provider class implements what is conceptually a set of related types. One attribute of the class functions as surrogate *type* information and is used in a decision structure to handle different variations of functionality required.

```
class Message {
public:
  Message();
  set_value(char* text);
  set_value(int action);
  void send(Channel c);
  void receive(Channel c);
...
private:
  void* data;
  int   type_;
}
// from Message::send
  const int TEXT   = 1;
  const int ACTION = 2;
  switch (type_) {
  case TEXT: ...
  case ACTION: ...  };
```

Final Situation.

The case statements have been replaced by polymorphism and the original class has been transformed into a hierarchy comprised of an abstract superclass and concrete subclasses. Clients must then be adapted to create the appropriate concrete subclass.

Initially there may be a large number of dependencies on this class, making modification expensive in terms of compilation time, and increasing the effort required to test the class. The target structure improves all of these problems with the only cost being the effort required to refactor the provider class and to adapt the clients to the new hierarchy.

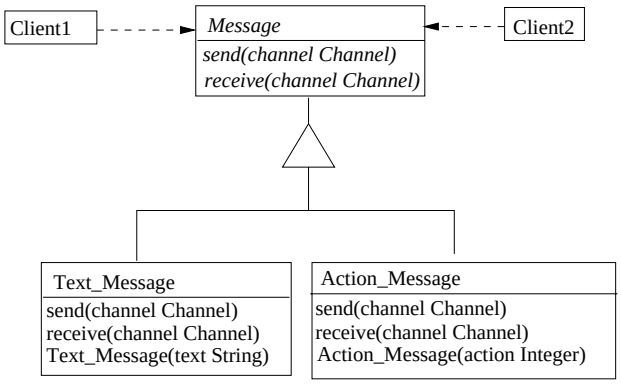


Figure 11.4: Final relation and structure of clients and providers.

```
class Message {
public:
  virtual void send(Channel c) = 0;
  virtual void receive(Channel c) = 0;
...
};

class Text_Message: public Message {
public:
  Text_Message(char* text);
  void send(Channel c);
  void receive(Channel c);
private:
  char* text;
...
};

class Action_Message: public Message {
public:
  Action_Message(int action);
  void send(Channel c);
  void receive(Channel c);
private:
  int action;
...
};
```

Structure

Participants.

- A single **provider** (Message) class that is transformed into a hierarchy of classes (Message, Text_Message and Action_Message)
- A set of **client** classes

Collaborations. The single provider class will be transformed into a hierarchy, thereby increasing modularity and facilitating extension of functionality.

Initially, the clients are all dependent on a single provider class. This class encompasses several variants of functionality and thus encapsulates all the collaboration that would normally be handled by polymorphism. This results in long methods typically containing case statements or other large decision structures.

The situation is improved by refactoring the single provider class into a hierarchy of classes: an abstract superclass and a concrete subclass for each variant. Each of the new subclasses is simpler than the initial class and these are relatively independent of each other.

Consequences. The functionality of the hierarchy can be extended adding a new subclass without modifying the superclass. The increased modularity also impacts the clients who are now likely to be dependent on separate subclasses in the provider hierarchy.

Process

Detection.

A class having many long methods is a good candidate for further analysis. A line of code per method metric may help to narrow the search. If these methods contain case statements or complex decision structures all based on the same attribute then the attribute is probably serving as surrogate type information. In C++, where it is a good practice to define a class per file, the frequency of case statements in the same file can be also used as a first hint to narrow the search for this pattern.

Example: detection of case statements in C++ Knowing if the pattern should be applied requires the detection of case statements. Regular-expression based tools like emacs, grep, agrep help in the localisation of case statements based on explicit construct like C++'s switch or Ada case. For example, grep 'switch' ‘find . -name "*.cxx" -print‘ enumerates all the files with extension .cxx contained in a directory tree that contains switch. The grep facilities for grep are extended in agrep so it is possible to ask for finer queries. For example, the expression agrep 'switch;type' -e ‘find . -name "*.cxx" -print‘ extracts all the files containing lines having switch and type.

However, even for a language like C++ that provides an explicit case statement construct, detecting case statements based on explicit ifthenelse structures is necessary. The tools above are not well suited for such a task, since their detection capabilities are restricted to one line at a time. One possible solution is to use perl scripts - a perl script which searches the methods in C++ files and lists the occurrences of case statements can be found in the appendix.

Recipe.

1. Determine the number of conceptual types currently implemented by the class by inspecting the case statements. An enumeration type or set of constants will probably document this as well.
2. Implement the new provider hierarchy. You will need an abstract superclass and at least one derived concrete class for every variant.
3. Determine if all of the methods need to be declared in the superclass or if some belong only in a subclass.
4. Update the clients of the original class to depend on either the abstract superclass or on one of its concrete subclasses.

Difficulties.

- If the case statements are not all over the same set of functionality variants this is a sign that it might be necessary to have a more complex hierarchy including several intermediate abstract classes, or that some of the state of the provider should be factored out into a separate hierarchy.
- If a client depends on both the superclass and some of the subclasses then you may need to refactor the client class or apply the TYPE CHECK ELIMINATION IN CLIENTS pattern because this is an indication that the provider does not support the correct interface.

Discussion

During the detection phase one can find other uses of case statements. For example, case statements are also used to implement objects with states [BECK 94], [ALPE 98]. In such a case the dispatch is not done on object type but on a certain state as illustrated in the State pattern [GAMM 95], [ALPE 98]. Moreover, the Strategy pattern [GAMM 95], [ALPE 98] is also based on the elimination of case statement over object state.

In his thesis Opdyke [JOHN 93] discusses the automatisation of code refactoring. His "Refactoring To Specialise", in which he proposed to use class invariant as a criteria to simplify conditionals, is similar to this pattern.

Language Specific Issues.

C++ Detection: in C polymorphism can be emulated either by using function pointers or through union types and enum's. C++ programmers are likely to use a single class with a void pointer and then cast this pointer to the appropriate type inside a switch statement. This allows them to uses classes which are nominally object-oriented as opposed to unions which they have probably been told to avoid. The use of constants is typically favoured over the use of enum's.

Difficulties: If void pointers have been used in conjunction with type casts then you should check to see if the classes mentioned in the type casts should be integrated into the new provider hierarchy.

ADA Detection: because Ada83 did not support polymorphism (or subprogram access types) discriminated record types are the preferred solution. Typically an enumeration type provides the set of variants and the conversion to polymorphism is straightforward in Ada95.

SMALLTALK In SMALLTALK the detection of the case statements over types is hard because few type manipulations are provided. Basically, methods isMemberOf: and isKindOf: are available. anObject isMemberOf: aClass returns true if anObject is an instance of the class aClass, anObject isKindOf: aClass returns true if aClass is the class or a superclass of anObject. Detecting these method calls is not sufficient, however, since class membership can also be tested with self class = anotherClass, or with property tests throughout the hierarchy using methods like isSymbol, isString, isSequenceable, isInteger.

Tools

Glimpse and agrep can be found at ftp://ftp.cs.arizona.edu/glimpse.

Known Uses

In one FAMOOS case study several instances of this problem were found. In the example studied in depth (DialogElement) it appears in conjunction with a class that groups together user interface and core model functionality. There is a data member called _type that is used in the various switch statements. Furthermore a void pointer is frequently cast to an appropriate type based on the value of _type.

Chapter 12

Cluster: Duplicated Code

DETECTION OF DUPLICATED CODE

Author(s): Matthias Rieger and Stéphane Ducasse

Intent

Detect code duplication in a system, without prior knowledge of the code. Identifying the duplicated code is a first important step towards application refactoring.

Applicability

The only prerequisite is the availability of the source code.

Symptoms.

- You already saw the same source somewhere else in the application.
- You already fixed the same error in another piece of code.
- You make a conceptual change and in adapting the software to the new concept have to edit similar pieces of code over and over again.
- You know you employed copy and paste programming during development, but do not remember exactly were it was.

Reengineering Goals. Some of the following reengineering goals are not only linked to the identification of duplicated code but also to its removal by refactoring:

Identifying unknown duplicated code. This pattern is well-suited to identify **unknown** and middle size (4 to 100 lines) of duplicated code. If you are looking for occurrences of a particular line of code, use sed- or grep-like tools or emacs (regexp and etag) facilities. If you are sure that the developers had to use copy and paste coding (e.g. your software contains about 4 millions lines of code and was developed by 2 people during one year) but want to know what has been copied and pasted, apply this pattern.

Identifying duplicated code in large scale system. Following the previous point, if you are looking for a way to identify duplicated code in a big (100'000 lines) to huge system apply this pattern.

Improving maintenance. Detection helps the maintainer of a system to make sure that some code fragment, where an error has been fixed, is not copied a number of times with the error still in it, or, complicating matters further, is fixed differently at each location by maintainers who have no knowledge of each other's activities.

Reducing maintenance cost. By detecting clones of a piece of code to be maintained and merging the code into one instance, the multiplied effort otherwise necessary to maintain all the clone instances is removed.

Improving the code readability. By identifying duplicated code and refactoring it, the size of code is reduced. The level of abstraction is elevated when similar code pieces are refactored in a new method, ultimately leading to the SMALLTALK ideal of 6 lines of code per method. In one of the FAMOOS case studies, we found a method of 6000 lines of C++ code, which is a nightmare in complexity by any standards.

Improving compilation time. The less lines of code you have, the faster your system is compiled.

Reducing the footprint of the application. The less lines of code you have, the smaller the executable of your application gets.

Related Reengineering Patterns.

- The CUT AND PASTE anti-pattern [BROW 98] explains what practices lead to code duplication. The pattern discussed here focuses on the *detection* of the duplicated code.
- Patterns describing the factoring and reorganisation of code within the class hierarchy or by creating new classes. Such patterns detail how the detected clones can be merged into a single instance.

Motivation

The duplication of code occurs frequently during the development phase when programmers reuse tried and tested code in a new context, but are reluctant or, due to severe time pressure unable, to invest the time necessary to generalise the existing code to be used in the old and the new context. Since duplication is an *ad hoc*/copy&paste activity more than something that is planned for, occurrences of duplication are not documented and have to be detected.

Process

In order to detect code duplication in an unknown system, one cannot search for specific patterns. Rather, the self similarity of the system has to be discovered. Each copy is equal or similar to its clones and this similarity is revealed by comparing the entire system to itself. This comparison is on the one hand computing intensive and on the other hand produces a remarkable amount of data of possibly copied code pieces. It is therefore necessary to automatically narrow down the candidates that have to be examined in detail by a human.

Recipe. The applicability of the recipe is based on the availability of a tool for duplication detection.

1. Start with an automatic search for clones. The tool should create a database of all locations where code duplication possibly occurred.
2. Deciding on the level or size of duplication that is interesting, filters are defined that remove the uninteresting candidates.
3. For each clone family (i.e. $n \geq 2$ copies of the same piece of code) that is left after the filtering step, a list of source code locations, possibly already with citations of the offending code pieces, is presented to the maintainer so s/he can decide on how to remove the duplication.

Note that the recipe in this pattern does not concern itself with the actual problems of refactoring the code.

Difficulties. The approaches used to compare actual pieces of code work on syntactical representations. Therefore, one cannot detect duplicated *functionality* that does not bear any syntactical resemblance.

Language Specific Issues.

Language dependency can stem from the parser that transforms the source code in the format that is used for comparisons by the tool. Depending on this format, the parser can be of variable complexity. For example, comparing the source code as text with only minimal transformations, e.g. removing comments and superfluous white space, only needs a very simple lexer, which keeps language dependency at a low level. Comparing abstract syntax tree of the source code, however, requires a full blown parser. The complexity of the first transformation step thus correlates directly with language dependency.

Tools

Tool support is vital for applying this pattern.

- We have implemented a SMALLTALK tool called DUPLOC (see Chapter 17), which is specifically aimed at supporting a visual approach of code duplication detection. At the moment, the tool uses textual comparisons only. It allows the user to compare source code file by file, enabling him to examine the source code by clicking on the dots. Noise filtering can be done by removing uninteresting lines.
- DUP [BAKE 92] is a tool that detects parameterised matches and generates reports on the found matches. It can also generate scatter-plots of found matches.
- DOTPLOT [HELF 95] is a tool for displaying large scatter-plots. It has been used to compare source code, but also filenames and literary texts.
- DATRIX [MAYR 96b] is a tool that finds similar functions by comparing vectors of source metrics.

Discussion

This pattern is valuable to apply if your system has the symptoms identified above or if your reengineering goals belong to the set of the mentioned reengineering goals. It is also advisable, though, to apply it as a precautionary measure in the maintenance process as a *code investment* [BROW 98]. If you plan to revamp an old system, duplication detection can help to plan parts of the effort.

Moreover, if your system should be migrated from one paradigm to another one—e.g. from COBOL to an object oriented language like SMALLTALK —and you suspect duplicated code, this pattern is valuable to identify which parts of the old system have been duplicated. Assessing the similarities and differences of the parts will also improve your understanding of the systems functionalities.

The approach that has been taken in the development of DUPLOC (Chapter 17) has the following advantages:

- It is lightweight: it does not use complicated algorithms like elaborate parsing techniques.
- It is visual: the human eye is built to detect configurations and this can be fully exploited with a matrix visual representation.
- It is language independent: Since we use textual comparison, the tool is language independent to a high degree and can be used for a number of languages without a change.

Technical. The algorithm that is used to compare the source lines determines what level of fuzziness is allowed to recognise a match. The simplest algorithm—which compares the source lines character per character—finds only exact matches. More complicated algorithms (see for example [BAKE 95]) can find *parameterised matches*. Parameterised matches point out the possibility to refactor code into a parametrisable function, where exact matches emphasise more the repetitive structures in the source code.

Known Uses

The pattern has been applied in biology research to detect DNA sequences [PUST 82]. In the context of software reengineering, the pattern has been applied to detect duplicated code in FAMOOS case-studies containing up to 1'000'000 lines of C++. It also has been applied to detect duplicated code in a COBOL system of 4 millions of line of code. The DUP tool [BAKE 92] has been used to investigated the source code of the X-Window system, and DATRIX has investigated multiple versions of a large telecommunications system, wading through 89 million lines of code all in all [LAGU 97]. DOTPLOT [HELF 95] has been used to detect similarities in man-files, literary texts and names from file systems.

Chapter 13

Cluster: Improving Flexibility

REPAIRING A BROKEN ARCHITECTURE

Author(s): Holger Bär and Oliver Ciupke

Intent

Detect and remove dependencies between packages of a system that aren't allowed according to the designated system architecture. These architecture breaking dependencies may prohibit the exploitation of the architecture's advantages and cause unexpected effects at maintenance work.

Applicability

This pattern is only applicable if the system to re-engineer should suit a certain architecture like *Model View Controller* (MVC), should be layered or should obey other documented restrictions concerning the dependencies between its packages.

To clarify the further discussion we note that a dependency between two classes located in different packages implies a dependency between the corresponding packages with the same direction.

Symptoms.

- If you are to carry out a change on the system which is supported by the system's documented architecture, e. g. replacing the top level package in a layered architecture or adding a view to the model in a MVC architecture, the effort is higher than expected. This is due to extra dependencies breaking the architecture and resulting in a cascade of changes to the rest of the system.
- Analyzing the system one encounters forbidden dependencies between packages, e. g. model classes depending on their visual representation in an MVC architecture.

Reengineering Goals.

- If the conformance to a certain architecture is proven or recovered the benefits of the architecture can be exploited, e. g. it's easy to add a new view to a model within the MVC architecture.
- Understandability: the dependency constraints of the architecture reduce the number of dependencies.

Motivation

The following example describes a typical three tier architecture for business applications with a user interface, application logic layer and database layer. The architectural restriction on the dependencies between these three packages is that the user interface may depend on the application logic which may depend on the database, but nothing more.

Initial Situation. In our example the application logic layer implements financial transaction management and offers a service named reportTransactions (from, to) to report the transactions for a certain period of time. Figure 13.1 shows the three packages and their dependencies. Evidently there's a dependency breaking the architecture from the application logic layer to the user interface: the call new ListOutput (reportList) to create a new output window for lists offered by the user interface layer. The reason for introducing this call instead of returning the resulting list to the user interface layer might be the fear of performance penalties.

In general reasons leading to a broken architecture are:

- Altering the system without having understood the architecture.
- A system architecture which seems to have performance penalties that can be overcome by breaking the architecture.
- Favoring "quick-and-dirty" instead of "nice-and-clean".

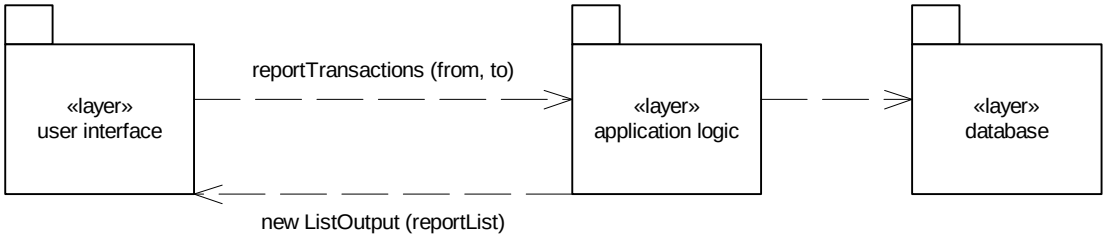


Figure 13.1: A broken architecture

Final Situation. The solution for the problem described above is quite straightforward. Just let the service computing the report return the transaction list to the user interface instead of itself displaying the list. But first it's not always that easy and second it's only easy after the dependency breaking the architecture has been found within the whole system.

Structure

There is no common problem structure for breaking an architecture because architectures themselves do not have a common structure. So a target structure is missing also.

Process

Detection.

1. Analyze the actual high level dependency structure, i.e. the dependencies between packages.
2. Search for dependencies which are not allowed by the documented architecture. You can do this

   Manually: Visualize the dependency graph with a graph layout tool and search manually for dependencies breaking the architecture.

   Automatically: The process can be automated with a tool that is able to analyze and manipulate graphs or relational queries on the given data.

(a) Set up a second graph containing the packages and the allowed dependencies between them according to the documented architecture.

(b) Compute the set of actual dependencies minus the set of allowed dependencies. The result are the dependencies breaking the architecture.

3. To find architecture violations in a system that should be layered search for cycles in the dependency graph.

Recipe.

A violating dependency exists either between two packages where no dependency is allowed or the dependency is just in the wrong direction like the one in our motivating example. In the first case there is no general solution, but in the second case the dependency can be reversed in a generic way:

1. Create a new abstract class with the same interface as the target of the dependency.
2. Replace the dependency on the target class by one on the new abstract class.
3. Let the target of the dependency inherit from the new abstract class. Now both the original source and target of the dependency are dependent on the abstract class.
4. Move the abstract class to a package where both the source and the target package may depend on it. In the case of reversing the dependency, this is the package containing the source of the dependency.

Figure 13.2 shows a broken architecture. Package P2 may depend on package P1, but P1 is not supposed to depend on P2. Actually class B depends both on classes C and D which makes P1 dependent on P2.

Figure 13.3 shows the solution to the problem. Instead of C and D, B calls now the abstract classes C_abstr and D_abstr. C and D inherit from their abstract counterpart and implement the methods called by B.

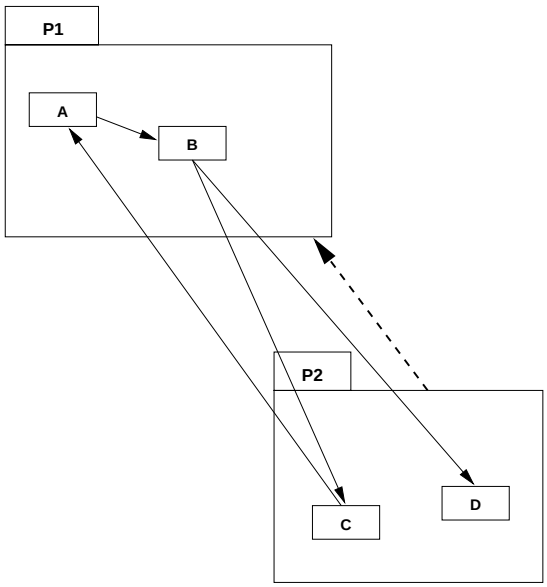


Figure 13.2: A broken architecture

There are also special solutions for similar problems like a model that has to update its various views without knowing how many views there are and of which type they are. This problem is solved by the OBSERVER pattern [GAMM 95].

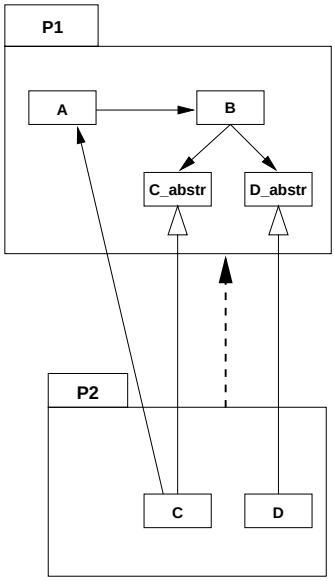


Figure 13.3: Dependencies reverted to fit the documented architecture

Difficulties. The new abstract class used by the source of the forbidden dependency B can be seen as an interface defined by B that supplier classes must implement. So this interface needn't contain all methods of the former target (C resp. D), but only the methods B needs.

In our example in the figures 13.2 and 13.3 there was only one class, class B, having a forbidden dependency on class C. If there is more than one class having a forbidden dependency the solution is a bit more complicated:

- Define only one interface per package that is used by all dependants on C.
- For dependants in different packages create one abstract class per package and let class C implement all of them or move the abstract class(es) into a package which every affected package may depend on.

Language Specific Issues. The transformation for reversing dependencies is only generally applicable for languages that allow an inheritance relation to be added to a pure abstract class (interface in JAVA). This is the case for C++ and JAVA and also for SMALLTALK , because in SMALLTALK there is no need to inherit from a pure abstract class.

Tools

Tools support is available for the following tasks of the detection section:

- Produce static structure graphs from source code.
  The tool set GOOSE contains too parsers, RETRIEVER and TABLEGEN with different advantages which can generate design information from C++ code in a format readable further tools.
- Visualizing a graph.
  You can use VCG, a graph layout tool, with a great variety of hierarchical layouts or Graphlet offering a set of layout algorithms with quite different aproaches. Unfortunately Graphlet (Version 2.8) has problems with printing the graphs.

- Setting up a new graph
  can be done with a graph editor like Graphlet.
- Finding cycles in a graph.
  Execute the command

  ```
  reView strongComponents < graph.gml | printCycles
  ```

  of the tool set GOOSE with graph.gml replaced by your graph file.
- Computing the difference between two graphs.
  The tool set GOOSE lets you convert graphs in a relational ASCII format. Filter off any other information besides first three collumns containing the type, source and target of the relation in these ASCII files with the Unix command cut -f1-3. Then use the Unix command comm -23 followed by the two files on which the difference should be computed.

Known Uses

In one of the FAMOOS case studies, there was an architecture defined with a base line framework and different products on top of this framework. When analysing the code, a class of the base line framework was found to inherit from several product classes. This kind of dependency was forbidden by the architecture definition. To repair this, an interface class was introduced from which the product classes inherited. This way, the framework was no longer dependent on the products, which made the system easier to change and decreased compile times.

A further example for an successful architecture clean up is the change in the event model of the Java Development Kit from the Version 1.0 to 1.1 (...cite). In this case the OBSERVER pattern was applied[1]. The observer pattern [GAMM 95] is a special form of the general principle to introduce an abstract interface to decouple classes.

[1] In the JDK, the Observer is called Listener.

TRANSFORMING INHERITANCE INTO COMPOSITION

Author: Benedikt Schulz

Intent

Improve the flexibility and comprehensibility of your design by transforming an inheritance relationship into a component relationship and delegating of a set of methods to this component.

Motivation

The following example occurred in a project which aimed at visualising hydraulic data of river parts. The data was visualised in a two-dimensional diagram which changed over time. The user of the system got the impression of seeing a film because of this animation.

The most crucial part in the system concerning efficiency was the subsystem which was responsible for drawing lines on the screen: For every new frame of the animation the complete set of lines representing the data had to be redrawn.

Initial Situation. In the first version of the system drawing lines was handled by the GDI subsystem of the Win32s operating system. This was pretty efficient until a new requirement came into play. The customers wanted to be able to change properties of the lines like colour, thickness, style, etc. The GDI subsystem was not able to draw lines with customisable thickness in an efficient way however: The system was showing rather a slide show than a film. The initial design is depicted in Figure 13.4.

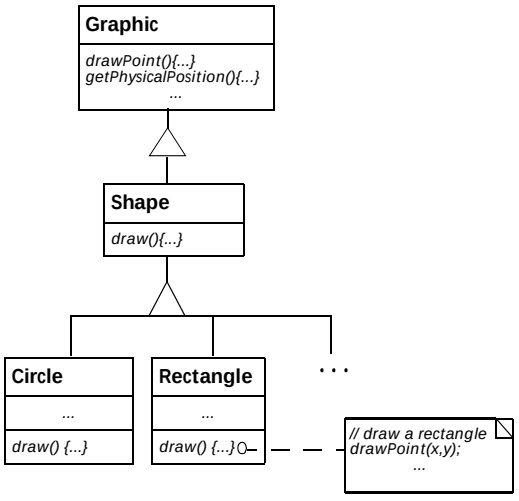


Figure 13.4: Initial Situation

Some experiments with a new technology called DirectDraw (that is also a subsystem of the operating system) revealed its superiority and thus the project manager decided to replace GDI with DirectDraw.

This led to serious problems: Since the class responsible for drawing lines was using functionality of GDI by inheritance it was not possible just to replace it by DirectDraw. DirectDraw had a different interface and so the implementation of a lot of methods which were responsible for drawing lines had to be changed.

Final Situation. To avoid similar problems in the future the project manager decided not only to change the the drawing system but additionally to introduce a flexible new design which should allow for easy exchange of different drawing systems.

The new design got its flexibility mainly from one change: Instead of relying on inheritance to reuse functionality, a component relationship together with the concept of delegation was used. This means that a `Shape`-object no longer ”knows” (directly or via inheritance) how to draw points but it rather ”knows” an object which ”knows” how to draw the points. Since objects can even be changed during run-time of the system the flexibility of the system was significantly improved. The final design is depicted in Figure 13.5 where new or changed entities are marked grey.

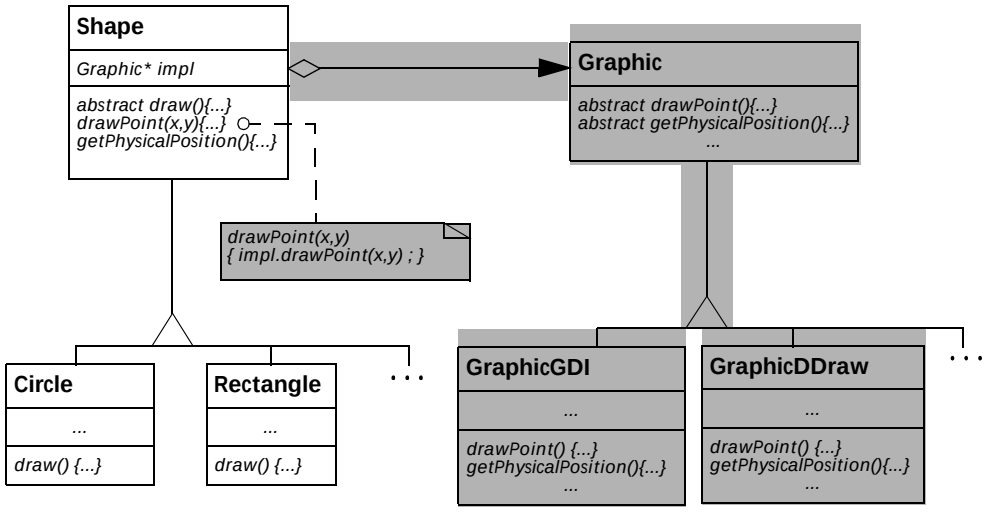


Figure 13.5: Target Structure

Some weeks after the redesign of the system it was revealed that the DirectDraw subsystem was not automatically installed on all systems running Win32s. But since the system could check whether DirectDraw was installed or not during run-time and since the drawing system was made exchangeable during run-time this new fact did not lead to any problems.

In the end the target structure is an instance of the Bridge design pattern [GAMM 95]. (It was not possible to use a Singleton Graphic acting as a facade to the libraries, because Graphic is not stateless and can have different states for different Shape-objects.) The Transforming Inheritance into Composition pattern is nevertheless not equivalent to the Bridge design pattern, because it not only describes “good” target structures but rather the process of applying the Bridge design pattern to an existing object-oriented legacy system.

Applicability

Transforming Inheritance into Compositionis applicable whenever you recognise during the reviewing of your legacy system that *you should have used* one of the following design patterns *but you have not used* them:

- Bridge [GAMM 95],
- Strategy [GAMM 95] or
- State [GAMM 95][DYSO 96].

All of these design patterns make use of the Objectifier design pattern [ZIMM 95] and the technique of delegation.

The application of this pattern is difficult if the inheritance relationship is deeply nested in the hierarchy because breaking the hierarchy means that all the methods which were inherited (and this can be a large number) have to be delegated. Therefore the inheritance relationship is *not* removed in a variant of the Transforming Inheritance into Composition reengineering pattern which will be discussed later.

The reengineering pattern should not be used in the following cases:

- Inheritance *is* the appropriate modelling technique for the problem (e.g., if there is a *is-a* relationship between two classes).
- Introducing delegation would be too expensive with respect to efficiency. This has to be considered especially when the delegation takes place within a loop which is processed a lot of times.
- In statically typed languages: Clients use the two classes related via inheritance polymorphically and you do not want to change these clients.

Symptoms. The application of this pattern can improve your design if you encounter one of the following problems:

- For a certain problem you should have used the *Bridge*, *Strategy* or *State* design pattern but in the system you are reengineering these design patterns have not been used. You know how to use the respective design patterns when you are building a new system but you do not know how to apply them to an existing design.
  - You want to be able to change the implementation of an abstraction in a more flexible way, maybe even at run-time (*Bridge* design pattern). The actual design does not allow for this kind of flexibility.
  - You want to extend the class system with new classes which share the same interface but differ in their behaviour (*Strategy* design pattern). The actual design does not allow for this kind of flexibility.
  - You have a lot of conditional statements in your code because the behaviour of an object depends strongly on its current state. You want to get rid of these conditionals (*State* design pattern).
- The inheritance relationship was established mainly for code reuse. The code which was the reason for using inheritance now has to be changed and so you want to remove the inheritance relationship because it is no longer appropriate. You do not know how to do this without changing the functionality of the system.

Reengineering Goals. The goal of the Transforming Inheritance into Compositionreengineering pattern is to help software engineers to apply a design pattern relying on the Objectifier design pattern and delegation to an existing design. In particular the pattern aims at

- increasing run-time flexibility. This is achieved because after the application of the reengineering pattern you will be able to change the component during run-time.
- increasing static flexibility (configurability). This is achieved because after the application of the pattern you will be able extend the component class hierarchy independently from the abstraction.
- increasing comprehensibility. This is achieved because the reengineering pattern can remove inheritance for code reuse which is hard to understand from your system.

Related Reengineering Patterns. The Transforming Inheritance into Compositionreengineering pattern is related to all design patterns which rely on the Objectifier design pattern [ZIMM 95] and delegation like

- Bridge
- Strategy
- State

Structure

The problem structure is depicted in Figure 13.6. Transforming Inheritance into Compositionleads you to the target structure depicted in Figure 13.7

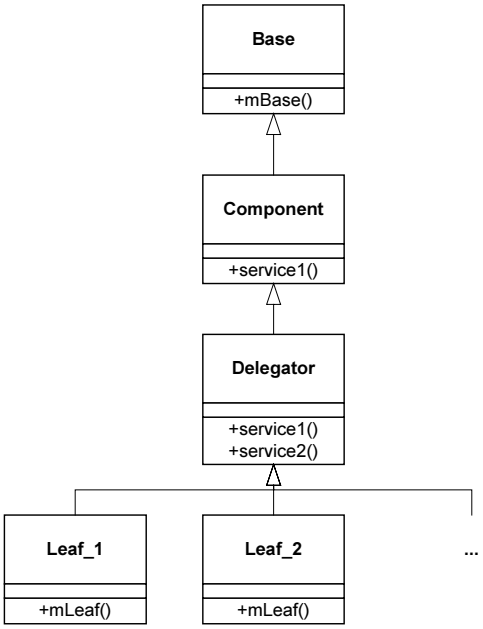


Figure 13.6: Problem Structure for the reengineering pattern

Participants.

- **Base** is the root of the inheritance tree.
- **Component** (Graphic) is the class which gets cut out from the inheritance hierarchy to serve as a provider of certain services. The inheritance relationship to `Base` may remain in existence.
- **Delegator** (Shape) is the class which uses services from `Component` by inheritance in Figure 13.6. After application of the reengineering pattern in Figure 13.7 `Delegator` will make use of these services by delegation.
- **Leaf_1, Leaf_2, ...** (Circle, Rectangle, ...) are the leaves of the inheritance hierarchy
- **Component_A, Component_B, ...** (GraphicGDI, GraphicDDraw, ...) are the subclasses of `Component` implementing the services of their super-class in different ways..

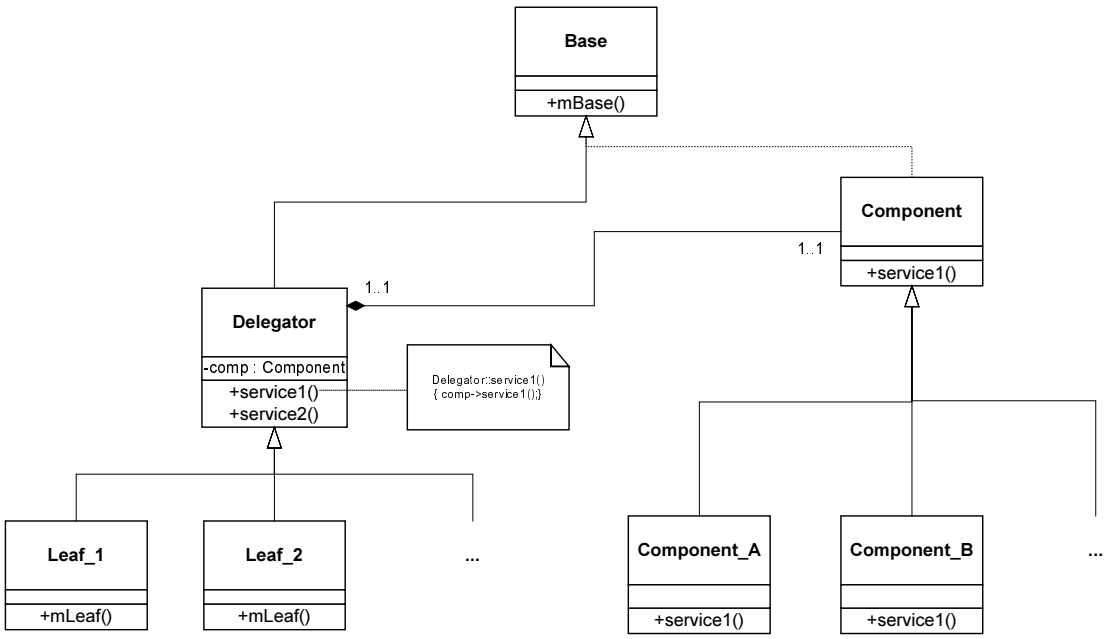


Figure 13.7: Target Structure for the reengineering pattern

Collaborations.

- `Delegator` makes use of **service1** (drawPoint) provided by `Component`. This is done
  - in the problem structure by executing inherited methods from `Component` whereas
  - in the target structure the execution of these methods is *delegated* to `Component`.

Consequences.

- Positive benefits
  - Transforming Inheritance into Compositionsolves an important and basic reengineering problem and the application of the reengineering pattern allows for the introduction of several known design patterns [GAMM 95].
  - Since abstraction and implementation are separated, changing the implementation does not require recompilation but only rebinding of the system.
  - The implementors of `service1` can be designed to form a separate inheritance tree. (This is suggested by the class `ComponentA` in Figure 13.7.) This is impossible before the application of the reengineering pattern.
- Negative liabilities
  - The execution of `service1` provided by `Delegator` will take longer in the target structure because it has to be delegated. This may be critical if `service1` is needed a lot of times.
  - The target structure is slightly more difficult to implement since the attribute of `Delegator` named `comp` has to be initialised whenever a new instance of `Delegator` is created and destroyed whenever that instance is deleted.

Process

The process mainly relies on the idea of combining the approach of considering design patterns as operators [ZIMM 97] (rather than building blocks) and the refactoring approach presented in [JOHN 93]. This idea is presented and discussed in detail in [SCHU 98b].

Detection.

Since violations against flexibility issues can only be detected if you know where flexibility is needed and which kind of flexibility (e.g., run-time flexibility, configurability) is needed, algorithmic detection is difficult. However, you can

- ask people who designed and implemented the system if there is a case where they wanted to be able to change the implementation of an interface at run-time and this was not possible.
- look for methods with a large amount of conditional statements. The behaviour of an object may depend strongly on its internal state (Type Check Elimination within a Provider Hierarchy).
- look for two classes, one inheriting from the other, which are never used polymorphically. This means that a variable declared as super-class is never used for an instance of the subclass.

Recipe. In this section we show how to apply tTransforming Inheritance into Composition and what kind of reengineering operations have to be applied. If we name entities (like classes, methods and attributes) we refer to the participants of the problem structure depicted in Figure 13.6 and the target structure depicted in Figure 13.7.

1. Create a new attribute `comp` of `Component` in the class `Delegator`. Change the constructor method of `Delegator` so that it initialises the attribute `comp` with a new instance of `Component`. If you plan to add several subclasses of `Component` later on (you should do so!) than add a new formal argument to the constructor method of `Delegator` which will serve as an indicator of which concrete subclass of `Component` to use.
2. Copy all the signatures of the methods from `Component` which are visible to `Delegator` to `Delegator`. For each added method add an implementation which delegates the execution of the method to the corresponding method of `Component`. For an example, see the implementation of `Delegator:service1()` in Figure 13.7.
3. Remove the inheritance relationship between `Component` and `Delegator`. Caution: In statically typed languages you will not be able to use an instance of `Delegator` polymorphically as an instance of `Component` after this step. In particular it is not possible any more to cast instances of `Delegator` to `Component`.

Difficulties.

If you decide to introduce an additional formal parameter to the constructor of `Delegator` then every piece of code that creates an instance of `Delegator` has to be changed. In languages which support default values for formal parameters this problem can be resolved by defining an appropriate default value (e.g., `Component` if this class is not made abstract).

If there is no way to avoid polymorphism between `Delegator` and `Component` but you still have strong reasons to apply Transforming Inheritance into Composition and you are using a statically typed language, you can omit removing the inheritance relationship between `Component` and `Delegator`. You should be aware of the fact, that you might have the following problem: The class `Component` has two parts:

- One part of the methods represents set of utility services. You made `Delegator` inherit from `Component` because you wanted to be able to use these services without re-implementing them.
- The other part of the methods represents the *real* interface of `Delegator`. You made `Delegator` inherit from `Component` because you wanted to establish an *is-a* relationship between `Delegator` and `Component` to be able to use instances of both classes polymorphically.

In this case consider splitting the `Component` class into two separate classes.

Language Specific Issues.

- In C++ you should implement the attribute `comp` as a pointer. Otherwise you will not be able to use polymorphism for the inheritance tree with root `Component`.
- In dynamically typed languages like SMALLTALK it is not necessary that two classes are related via an inheritance link to use them polymorphically. This means, for example, that you can still use instances of `Component` and `Delegator` together in one container object.

Discussion

Since the detection of the problem structure is far away from being an algorithmic, tool supported process, you should not explicitly look for this problem structure. But since software development is an iterative process you will find the problem structure while trying to extend or modify your system. Once you have found the problem structure in your code, you should strongly consider the application of Transforming Inheritance into Composition.

The relevance of this reengineering pattern is high: In a lot of companies which were early adopters of the object-oriented paradigm, the maturity of the software engineers concerning object-oriented technology was low. This resulted in an overuse of inheritance, mainly for code reuse. These software defects can be removed by the application of the reengineering pattern.

The concept of delegation and the Objectifier design pattern [ZIMM 95] are the fundamentals of this reengineering pattern and the resulting target structure is closely related to the Bridge, Strategy and State design patterns [GAMM 95]. A good understanding of these design patterns helps to use the reengineering pattern.

Tools

The detection of pairs of classes which are never used polymorphically can be done with the tool-set *Goose* [BÄR 98][CIUP 99]. *Goose* can not only detect missing polymorphism but a lot of other design defects which occur in object-oriented systems.

Since the application of the reengineering pattern relies on the application of refactorings [OPDY 92] you can use every tool which supports this technique, such as the *Refactoring Browser* [ROBE 97a] for SMALLTALK , which is the most advanced refactoring tool. The Refactoring Browser is described and available for free at `http://st-www.cs.uiuc.edu/~brant/Refactory/`.

For a subset of C++ we implemented a prototype to support refactorings. This tool is called *RefaC++* and described in [MOHR 98]. *RefaC++* can perform a subset of the refactorings presented in [OPDY 92] and can also apply the Bridge design pattern automatically.

Known Uses

Transforming Inheritance into Compositionhas been applied in the following known cases:

- The reengineering pattern was applied with success in the project described in the motivation section. It was possible to increase the flexibility of the system so that the new requirement (DirectDraw not available on every Win32s installation) could be fulfilled without problems.
- We are analysing and flexibilising a graphical information system for a German middle-sized enterprise. We found several design flaws which have been corrected by applying this reengineering pattern.
- [ROBE 96] describes how frameworks evolve. In the White-box Framework design pattern [ROBE 96] the engineer is encouraged to use inheritance for reuse because it is easier to understand and to reuse. In later stages of the framework development inheritance has to be replaced by polymorphic composition.

DISTRIBUTE RESPONSIBILITIES

Author(s): Holger Bär and Oliver Ciupke

Intent

Distribute the responsibilities equally among the classes of an object-oriented system to prevent large, hardly maintainable and reusable classes.

Applicability

A responsibility is a description of a service offered by a class. It is fulfilled by a set of publicly accessible methods.

Symptoms. If the responsibilities aren't distributed among the classes, there will be one or more classes incorporating a lot of responsibilities. Such classes, called *multiple responsible classes* (MRC) from now on, result in the following symptoms.

- If you ask for the responsibilities of a MRC, you get long and unclear answers.
- The MRC is used by other classes for different purposes (low level MRC).
- The MRC uses a lot of classes (high level MRC or *manager class*).
- Functional enhancements somewhere in the system often require changes in one of the high level MRCs.
- A MRC is mostly large in lines of code and number of methods, because many responsibilities result in many methods resulting in many lines of code for concrete classes.
- High level MRCs can hardly be reused because too many design decisions of the specific application are coded into them.
- Maintenance work on MRCs is hard, because there is no boundary between the different responsibilities, so that it's unclear where to change the class for a certain maintenance action and which the effects of the change are.

Reengineering Goals.

- Understandability: classes with many responsibilities are hard to understand, because the responsibilities are mixed together, i. e one can't identify the individual responsibilities and understand their implementation and collaboration with other classes in isolation.
- Flexibility: classes with few responsibilities allow fine grained adoptions by subclassing or replacing a class.
- Reusability: classes are normally reused as a whole. Therefore it's unlikely that MRCs are reused because the particular combination of responsibilities needed in its original application is unlikely to occur in another one.

Related Reengineering Patterns.

- large classes
- large methods
- structural programming
  Classic structural programming principles applied to an OO-language often lead to one central manager class operating on several dumb data classes.

Motivation

Initial Situation. The UML diagram in Figure 13.8 shows a manager class, AccountManager, together with two passive classes, AccountData and BarChart. The responsibilities of the manager class are

1. to process user input in OnCalculateSummary,
2. to do the summary calculation using Get...Transaction methods to query the transaction data from class AccountData,
3. and to present the results with the help of class Screen.

So the manager class has three responsibilities and implements nearly the whole functionality. Major design decisions like how the summary is calculated and presented and the reactions on the user input are hard coded in this class thus making it hardly reusable.

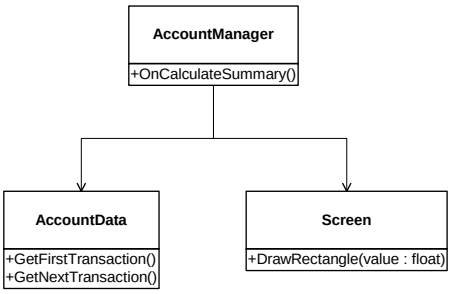


Figure 13.8: Example of a manager class with two passive classes.

Final Situation. We can distribute the three responsibilities of the manager class among three classes: UserInteraction, Account and BarChart. In this design all classes besides UserInteraction have a high potential for reuse.

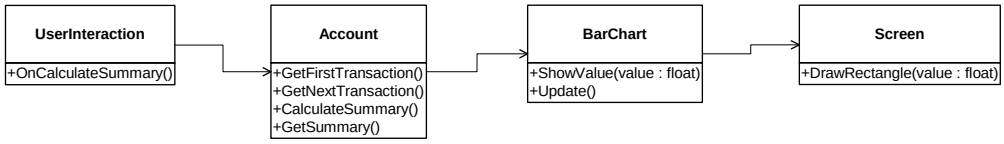


Figure 13.9: The improved example with distributed responsibilities.

Structure

The structure of the problem and the target structure differ both between high level MRCs and low level MRCs.

High level MRC problem structure.

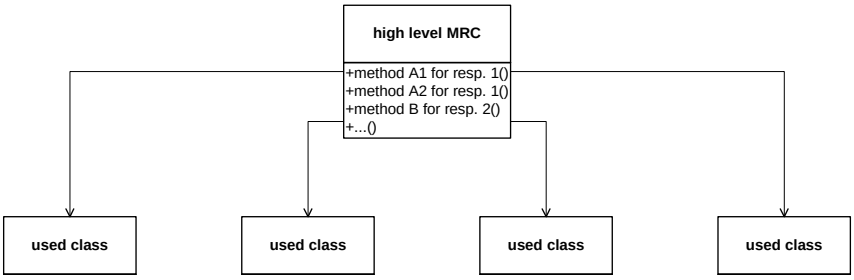


Figure 13.10: Problem structure of a high level MRC.

Participants. The high level MRC shows a broad interface with a set of methods per responsibility.

Collaborations. High level MRCs often use many other classes to fulfill their numerous responsibilities.

Low level MRC problem structure.

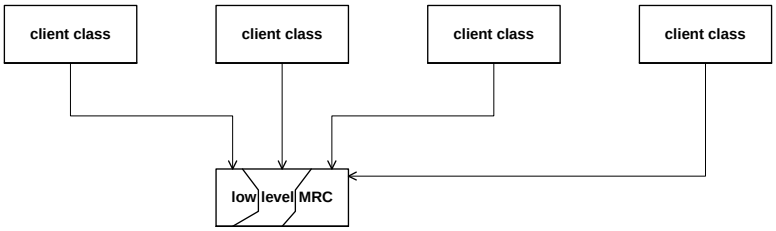


Figure 13.11: Problem structure of a low level MRC.

Collaborations. The various clients use different responsibilities of the low level MRC.

High level MRC target structure.

For high level MRCs there is no general target structure. The goal is to distribute the responsibilities. Good candidates for receiving responsibilities are the used classes. But sometimes it's necessary to define a new class like BarChart of the motivating example. The manager class itself will be reduced in size (lines of code, methods) or will disappear completely.

Low level MRC target structure.

Participants. The low level MRC has been split into several classes according to the responsibilities of the MRC and the parts used by the client classes.

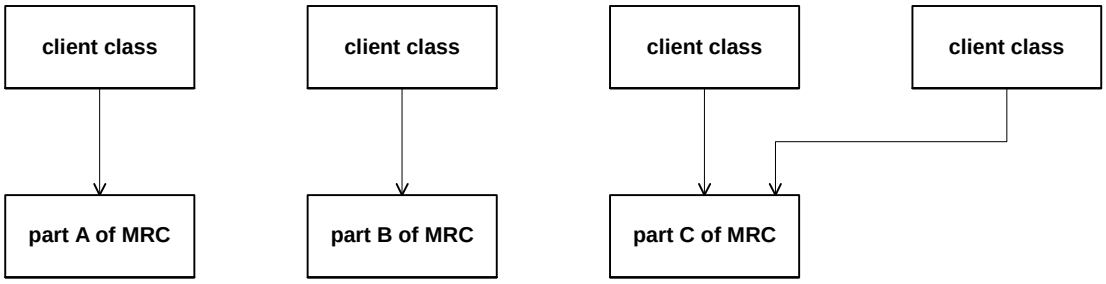


Figure 13.12: Target structure of a low level MRC.

Collaborations. The client classes of the low level MRC now only use those parts of the MRC they actually need.

Consequences.

- The responsibilities of the high level MRC are distributed moving the code closer to the data it works on.
- In case the manager class disappears completely there is no central control any more. The design has made a step towards autonomous interacting objects.
- The remainder of the high level MRC and the parts of the low level MRC are smaller, easier to understand and exhibit more potential for reuse.
- Instead of many dependencies on one low level MRC the application of this pattern leads to a set of classes each with a lower number of dependents.
- The smaller ”part classes” of the low level MRC are more stable than the original class simply because they encapsulate less design decisions. So together with the previous topic the compilation times after changes to the system will be reduced.
- In both cases the number of classes may increase.
- The distribution of responsibilities may affect the efficiency of the system.

Process

Detection.

- MRCs are normally the largest classes in a system, program or package both in lines of code and in number of methods.
- To find high level MRCs search for classes with manager, man, driver, initiator and so on in their name.
- Classes that use many other classes are also good candidates for high level MRCs. They can be found by looking for classes with high values for coupling metrics like the CBO metric [CHID 94].
- Low level MRCs are used for different purposes. So the implementation of their responsibilities are likely to not communicate with each other. Therefore these classes will often exhibit low cohesion. There are numerous cohesion metrics, e. g. the TCC metric [BIEM 95].

The conjunction of the size and coupling criteria and optionally the name criterion should produce satisfying results for the detection of high level MRCs as should the conjunction of the size and cohesion criteria for low level MRCs.

Recipe.

- Search for candidate MRCs as described in the detection section above. The next steps depend on whether you have detected a high level or a low level MRC.
- High level MRCs
  Try to distribute the responsibilities of the manager class to other classes. Good candidates for receiving responsibilities are the classes used by the manager class. It may be necessary to define a new class like BarChart of the motivating example. The manager class itself will be reduced in size (lines of code, methods) or will disappear completely.
- Low level MRCs
  1. Determine the parts of the low level MRC.
     There are two ways to determine the parts — one considers the use of the class by its clients, the other one examines the class' internal structure:
     (a) Analyse the use of the class by its clients. Note for each type of client the features of the MRC (methods and public attributes) it uses. Find a partition of the feature set so that each client uses only one or few parts.
     (b) Although cohesion metrics indicate whether a class could and should be split, they do not directly indicate where to split the class. You can get good suggestions for splitting by computing the minimum cut on the undirected graph containing all methods and attributes of the class as nodes and all method calls and variable accesses within the class as edges[2]. The minimum cut algorithm computes such a partition of the graph in two sets of nodes that the number of edges from one set to the other is minimised. Splitting a class according to this partition leads to two classes with minimal communication between them. Of course this splitting step can be applied to the two sets recursively until the resulting classes are small enough.
  2. You may use one of the mentioned partitions of the MRC or a combination of both to split the MRC. In cases where there is no optimal partition (e. g. client uses more than one part or there is more than zero communication between the parts) the partition often needs some manual fine tuning to end up with a set of reasonable classes.
  3. Split the MRC according to the partition and reorganise its context.

Difficulties.

- The detection of MRCs is not very precise, especially for low level MRCs. The key point in detecting a low level MRC is to recognise that it is used by different types of clients for different purposes.
- Determining the partition of the MRC can hardly be done fully automatically because the parts must be reasonable classes.

Tools

Several prototype tools have been developed within the FAMOOS project which can help to detect and to solve this problem.

[2] All edges have weight 1; parallel edges are allowed for modelling multiple calls and accesses

- A visualisation of the static system structure at the right level of abstraction, e.g. with the tools within the tool set GOOSE can help detecting central classes or subsystems.
- TABLEGEN computes the TCC cohesion metric and also coupling metrics.
- The computation of minimum cuts to determine the partition of a MRC can be done with REVIEW, a tool also developed within the FAMOOS project.
- GOOSE' relational representation of design information enables to search for classes with a high out-degree of usage. The classes with the highest out-degrees are good candidates for being high level MRCs.

Known Uses

During restructuring in one of the FAMOOS case studies, there was a big class found which incorporated responsibilities for several different products. This class was split into several pieces to make the program more flexible with respect to frequent changes [RITZ 98].

USE TYPE INFERENCE

Author: Markus Bauer

Intent

It is hard to understand the structure and the workings of a software system written in a dynamically typed language because of the lack of type declarations. Therefore add type annotations to the program code which document the system and which can additionally be used by sophisticated reengineering tools.

Applicability

Apply this pattern when reengineering systems that are written in *Smalltalk* or in a similar, dynamically typed programming language, where you have only limited knowledge about the system. Typical situations could be:

- You have to maintain and/or modify the software system, but you have only limited knowledge about its inner workings. You are interested to learn, which types of objects of the system are manipulated by some code your are working on, but this is difficult since you do not have type declarations in your source code that provide you with that information.
- You want to support a reengineering task by some tools, but these tools rely on type information for the system's variables and methods. Most reengineering tools rely on such type information. Examples include (but are not limited to) the *Smalltalk Refactoring Browser* [ROBE 97a][3] or tools that calculate software product metrics (like those described in [CHID 94]).
- You want to reengineer or rewrite the system using a statically typed programming language, but to achieve this, you need appropriate type declarations for the system's variables and methods.

Problem

In dynamically typed systems, the lack of static type information (i.e. the lack of type declarations for variables and method signatures) makes some reengineering tasks difficult or impossible, since such type information usually represents prominent parts of a system's semantics.

Motivation

Consider some code fragments[4] for a dynamically typed application that manipulates drawings. Such an application might have a class `Container` for storing some objects. Figure 13.13 shows a method `add` that is used to add objects to the container.

For reengineering purposes we might be interested in an answer to the following question: What kind of objects can be stored in the container, that is, of what types are the objects, that are passed as arguments to the `add`-Method?

[3] The current implementation of the Refactoring Browser does not infer precise types for the system's entities though, it relies on (unprecise) heuristics instead.

[4] We present code examples in a syntax close to Java. Since we deal with dynamically typed code, we just omit Java's type declarations.

```
add: anObject
   contents add: anObject.
   anObject draw.
   "..."
```

Figure 13.13: Method `add` in class `Container`.

Forces

- To learn about the types of objects that are manipulated by some code you are looking at, you might consider manually tracing the execution of your code and guess what's going on in your system, but for larger systems, this is an infeasible and error-prone task.
- You could also try to capture that information by looking at method and variable names, but in many legacy systems naming conventions do not exist or do not provide enough information about the object types and the manipulations that are made with them (see our example above). Even worse, you can't be sure that the names do not lead you to wrong conclusions.
- To migrate from a dynamically typed language to a statically typed language, you could apply approaches that do not rely on type information, like those proposed for the translation of Smalltalk applications to Java in [ENGE 98]. These approaches simulate Smalltalk's dynamic type system in Java. The resulting code, however, is not authentic Java code and is hard to understand and maintain. Such code additionally has the usual shortcomings of untyped code: it is not type safe.

Solution

Find out what types the variables and method parameters have and put this information into the source code, using type annotations or comments.

In more detail:

1. Perform a program analysis of your dynamically typed object oriented legacy system.
2. Use the results of the program analysis to determine type information for the program's variables, including global and local variables, parameters and return values of methods. Based on this type information, add type annotations to the program's source code.
3. Use these type annotations to understand how your legacy system works or as additional semantic information to more sophisticated reengineering tools.

This technique is called *type inference*, because you infer the type of an object at a certain place in the code by tracing its way from its creation to the current place.

If we can enrich the code of our example application with type annotations (see figure 13.14) by using the techniques described below we can easily find an answer to the question we asked above: Our `Container` holds points, lines, splines, ..., so it has obviously something to do with some geometrical shapes that make up a drawing.

We learn from this example that type annotations like those given in figure 13.14 make code much easier to understand and that they contain valuable information about the inner workings of a system.

```
add: anObject
   " {Container} × {Point, Line, Spline,...} → {} "
   contents add: anObject.
   anObject draw
   "..."
```

Figure 13.14: Method `add` annotated with type information.

Process

Type inference usually can't be done manually for reasonable large and complex applications. Therefore, we have to automate the task of computing type information for variables and method signatures.

To implement a tool or other means to get the information, we observe that during the runtime of the system, type information propagates through the system's expressions and statements: Upon creation, each object has a certain type assigned to it, and this type information is spread to all expressions and statements (including variable and method parameter expressions), that do some operations with the object. Thus, to infer types for the variables and methods of the system, we need to inspect object creations and the data flow through the system.

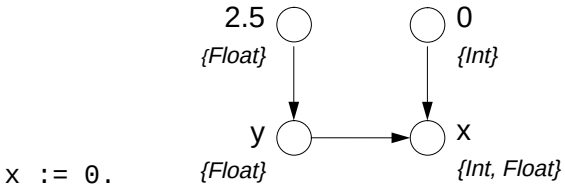
Basically we can do this in two ways: We either can execute the application and collect the type information we are interested in during its runtime *(dynamic type inference*) or we can use static program analysis techniques *(static type inference)* to analyse the applications source code and compute how the type information flows through the application's expressions. We will cover both approaches in some more detail below.

Dynamic type inference. With dynamic type inference, we modify the application or its runtime environment, to have it record the runtime type information for us.

1. Determine the most common execution paths through your program, that is, determine the most common usage scenarios of your legacy system. In some cases you might be able to use already existing testing scenarios for this. In other cases, determining these common usage scenarios might be difficult, especially if you don't know much about the system.
2. Instrument the code with instructions that record the data flow through your system and that collect the runtime types of the system's variables. [RAPI 98] describes how to modify the runtime libraries of a Smalltalk environment to achieve this with only minor changes to the application's code.
3. Run the system and have it execute the most common usage scenarios you collected in step 1.
4. Use the recorded runtime type information to put type annotations into the source code.

Static type inference. With static type inference, we need a tool that reads in the complete source code of the application and analyses it to construct a data flow graph. This is done by representing the application's expressions as nodes in the graph, and by modelling the dependencies between them as edges. The dependencies that are taken into account to construct the data flow graph are given by the following rules:

1. An *assignment* `var := expr` generates a data flow from the right hand side expression `expr` to the variable `var` on the left hand side.
2. A *variable access* generates a data flow from the variable being accessed to the surrounding expression.




```
x := 0.
y := 2.5.
x := y.
```

Figure 13.15: Data flow graph.

```
max: b
   (self < b)
      ifTrue: [^b]
      ifFalse: [^self]
```

Invocation of `max`:

```
x := 1 max: 2
```

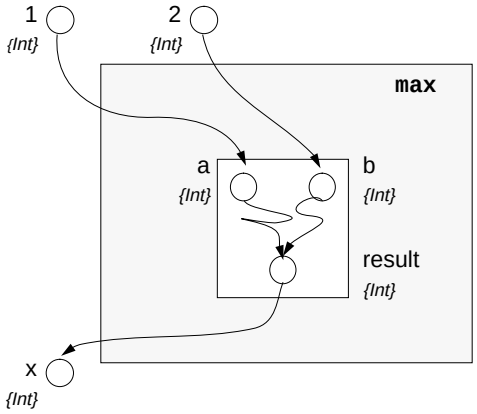


Figure 13.16: Data flow across method boundaries

3. A *method invocation* generates a data flow from the actual argument expressions to the formal arguments of the invoked method, and from the result of the invoked method to the invoking expression.

A data flow graph for a short piece of code is shown in figure 13.15.

For each node the tool then tries to compute the set of classes the corresponding expression can hold instances of. It starts by determining type information for the program's literal expressions and object creation statements (which are represented as source nodes in the graph) and moves that information along the edges through the graph. Each node then carries the union set of all type information of its predecessors. In figure 13.15, for example, the node for `x` carries the type information $\{Int, Float\}$, since it depends on the type information of the nodes for `y` and `0`.

Some subtle problems arise, whenever method invocations cause data flows across method boundaries (as given by rule 3). Such a case is shown in figure 13.16.

There are some well proven techniques to allow for an analysis which keeps track of these inter-method data flows in an efficient and practicable way. One of these is Agesen's Cross Product Algorithm [AGES 95][5].

[5]There are other algorithms that also allow the tracking of data flow across method boundaries, for example [PALS 91], [OXHØ 92], [PLEV 94], but Agesen's algorithm is superior to most of these, because it is easy to understand and computes precise type information in a very efficient way[AGES 94].

The basic idea is to create separate sub graphs for each method and link all those subgraphs together in an appropriate and efficient way.

After the graph has been complexly built up and all type information has been propagated through it, the type information associated with the graph's nodes can be used to annotate the source code of the application.

Discussion

A problem of using type inference to reveal some information about a legacy system arises from the fact that we analyse the data flow through an application. To make our approach work, we have to analyse the complete source code of an executable application (including libraries), or, if we are using dynamic type inference, we have to execute an adapted version of the system. This might be a problem in some cases when parts of the source code are not available and/or a runnable version of the system cannot be produced. Furthermore, frameworks and class libraries cannot be analysed without application code using or instantiating them. Then, however, the inferred types are only valid in the specialised context of the particular application.

Static type inference algorithms usually have to overcome some difficulties: static analysis is complex and the results are often unprecise. Agesen's static type inference algorithm, as sketched above, addresses these difficulties in an appropriate way[6]. However, since the algorithm is very complicated it is difficult to implement it in a correct way and produce a reliable tool out of it. This is an issue, if you can't use one of the already existing tools (see for example [LI 98]).

However, once a tool for performing such an analysis has been built, it can be used on other reengineering projects as well and then it quickly pays of its rather high development costs.

Dynamic type inference has serious limitations when being applied to larger systems: You have to ensure that the most important parts of the system are covered by the analysis in a sufficient way, which might not be feasible for larger systems if you do not have test cases or usage scenarios available.

Related Reengineering Patterns

Type annotations document the inner workings of a legacy system. We can therefore see type inference as a technique to improve your knowledge about the legacy system. Thus, this pattern relates with all other reengineering patterns that describe *reverse engineering techniques*, i.e. analyses of the source code of legacy systems to extract additional semantic information and improve the understanding of the systems.

Known Uses

ObjectShare has used type annotations (like those that can be computed by applying this pattern) to document large parts of the source code to the *Visualworks Smalltalk* environment. This emphasises that type annotations are of great help understanding source code.

The GOOSE tool set (and related tools) that support the reengineering of C++ applications by visualising software structures [CIUP 97], checking design heuristics [BÄR 98] and calculating software metrics [MARI 97] can analyse Smalltalk applications after type inference is used and the source code is enriched with type annotations.

The University of Stuttgart, Germany, has developed a tool called *Smalltalk Explorer* which is used to explore existing Smalltalk applications. It heavily relies on the type inference algorithm presented here.

[6] A detailed discussion of the algorithm, especially regarding complexity and precision can be found in [AGES 95] or in [BAUE 98].

Type annotations are used to allow for an easy navigation through unknown Smalltalk code by documenting which classes are manipulating which other classes and by introducing hyperlinks between them[LI 98].

The type inference algorithm is also used to facilitate a mostly automatic translation of dynamically typed Smalltalk applications into statically typed Java applications [BAUE 98]. Since most of Smalltalk's concepts can be mapped upon suitable Java concepts the most prominent issue is to infer appropriate static types for the resulting Java code. This is done by computing type annotations (as described above) and transforming them into type declarations. In more detail, to map a type annotation to a type declaration, a class must be found (or created by refactorings), that is a common abstraction to all classes included in the type annotation.

Part IV

Tools

Chapter 14

Tool support for reengineering

Author(s): S. Tichelaar and S. Demeyer

To be able to reengineer a legacy system successfully, tool support is needed. Tools help to cope with the vast amount of information normally found in legacy systems. They can provide a developer with different views on a system, point him to possible problems in the code, and help improve the software accordingly.

Within the FAMOOS project we have built tool prototypes to support us - and eventually developers - in performing reengineering tasks such as visualisation, metrics, heuristics and system reorganisation. All these tool prototypes have been tested extensively on the case studies of the industrial partners.

The tools presented in this part of the handbook are only the most mature tool prototypes developed in the FAMOOS project. The chapters contain descriptions of the functionality and scenarios how to apply the tools to solve reengineering problems.

- **AUDIT-RE (chapter 15, p. 221)** This tool integrates functionality developed in the FAMOOS project in the software quality assessment tool Concerto2/Audit.
- **GOOSE (chapter 16, p. 225)** A tool providing automated support for problem detection.
- **DUPLOC (chapter 17, p. 235)** A tool for detecting duplicated code.
- **CODECRAWLER (chapter 18, p. 241)** A tool that supports reverse engineering of large object-oriented projects by combining visualisation with metrics.
- **Nokia Reengineering Environment (chapter 19, p. 249)** A tool environment that integrates parsers, visualisation and metrics tools. It assists in the application of re-engineering methodology and automates basic and more repetitive operations.

To allow for integration of the different tools and to reuse tools for the different implementation languages of the case studies, the FAMIX model has been developed to support the exchange of information about object-oriented systems between tools (see also section 14.1). Within the FAMOOS project we are currently using this exchange model together with its representation in CDIF in our reengineering tool prototypes. Prototypes written in different languages work on sources written in different languages.

In the following section the FAMIX model is presented in more detail, and in the following chapters the different FAMOOS tool prototypes are presented.

14.1 The FAMIX Model

As legacy systems are written in different implementation languages (C++, Ada, Smalltalk and even Java), the tool prototypes need to be able to work with all of those languages. Therefore, we have developed a language-independent model for information exchange between reengineering tools, called the FAMOOS Information Exchange Model (FAMIX) [DEME 98b]. The FAMIX model provides for a language-independent representation of object-oriented sources and contains the required information for the reengineering tasks performed by our present tools. However, since we cannot know in advance all information that might be needed in future tools, and since for some reengineering problems tools might need to work with language-specific information (e.g. to analyse include hierarchies in C++), the FAMIX model must be extendible. Therefore, we allow for language plug-ins that extend the model with language-specific items. Secondly, we allow tool plug-ins to extend the model so tools can, for instance, store analysis results or layout information for graphs. Note that these plug-ins should not break language-independents tools.

Figure 14.1 shows schematically the use of the FAMIX model: the tools analysing the different languages and exchanging information with each other via FAMIX, possibly extended with language and tool plug-ins.

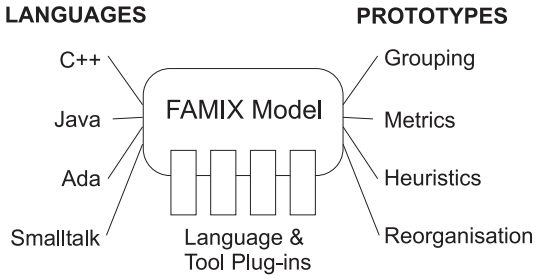


Figure 14.1: Concept of the FAMIX model

Using this model we don't have to equip all of the tools with parsing technology for all of the implementation languages. Next to that, the FAMIX model allows for exchange of information between the tools themselves. To exchange actual information, i.e. source code expressed in our model, we use the CDIF standard for information exchange [COMM 94].

The question may arise why we have defined our own model rather than extending an existing one such as UML. This issue is extensively discussed in [DEME 99d].

14.1.1 The Core Model

The core model specifies the entities and relations that are extracted immediately from source code (see figure 14.2).

The core model consists of the main OO entities, namely Class, Method, Attribute and InheritanceDefinition. In addition there are two associations, Invocation and Access. An Invocation represents the definition of a Method calling another Method and an Access represents a Method accessing an Attribute. These abstractions are needed for reengineering tasks such as dependency analysis, metrics computation and reorganisation operations. Typical questions we need answers for are: "are entities strongly coupled?", "which methods are never invoked?", "I change this method. Where do I need to change the invocations on this method?". The complete model consists of more information, i.e. more entities such as functions and formal parameters, and attributes for every entity, containing information about that entity. The complete specification of the model can be found in [DEME 98b].

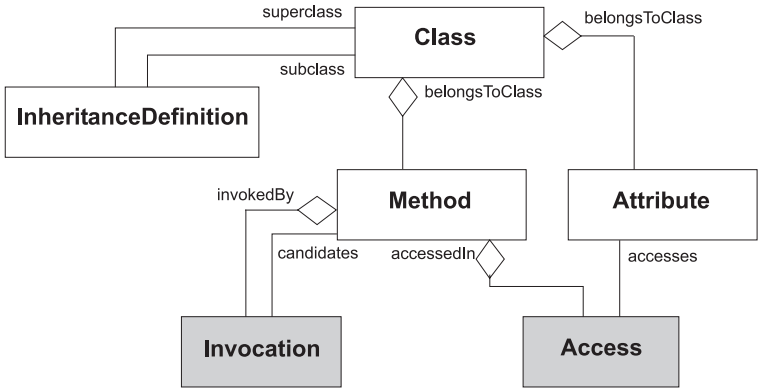


Figure 14.2: Core of the FAMIX model

14.1.2 CDIF Transfer Format

We have adopted CDIF [COMM 94] as the basis for the actual exchange of information using the FAMIX model. CDIF is an industrial standard for transferring models created with different tools. We have chosen CDIF as it perfectly matches the requirements we have for the representation of FAMIX-based information:

- *Easy to generate by available parsing technology* Since we cannot wait for future developments, we must use parsers available today, keeping an eye on short-term evolution.
- *Simple to process* As the exchange format will be fed into a wide variety of tool prototypes, the format itself should be quite easy to convert into the internal data structures of those prototypes. On top of that, processing by ”standard” file utilities (e.g, grep, sed) and scripting languages (i.e., perl, python) must be easy since they may be necessary to cope with format mismatches.
- *Convenient for querying* A large portion of reengineering is devoted to the search for information. The representation should be chosen so that it may easily be transformed into an input-stream for querying tools (e.g., spreadsheets and databases).
- *Human readable* The exchange format is used by (sometimes buggy) prototypes. To ease debugging, the format itself should be readable by humans. Especially, references between entities should be by meaningful names rather than by identifiers which bear no semantics.
- *Allows combination with information from other sources* Although most of the data model will be extracted from source code, we expect that other sources can provide input as well. In particular CASE tools like TDE or Rational/Rose are likely source candidates that, for instance, want to store and exchange design information. Thus, the representation should allow for merging information from other sources. Note that just like with the ”human readable” requirement this implies that references between entities should be by meaningful names rather than by identifiers which bear no semantics.
- *Supports industry standards* Since the tool prototypes must be used within an industrial context, they must integrate with whatever tools already in use. Ad hoc exchange formats (even when they can be translated with scripts) hinder such integration. When available, the representation should favour an industry standard.

Chapter 15

Audit-RE: a re-engineering tool

Author: Anne-Marie Sassen

During the FAMOOS project several tool prototypes have been developed. The aim of the prototypes was to test a certain functionality quickly. Therefore the effort spent to make them robust and easy to use was minimal. In order to solve this drawback, the best functionality of the prototypes has now been integrated or re-implemented in the software quality assessment tool Concerto2/Audit of Sema Group. The new toolset is called Audit-RE, and has the following functionality:

15.1 Robust parsing

Often, the only reliable documentation of a legacy system is the source code itself. The quality of the parser of the re-engineering tool is therefore of utmost importance. Pieces of code that are not parsed correctly cannot be re-engineered. Although parsing is a problem which seems to have been solved a long time ago, in practice many problems remain. During the case studies of FAMOOS, several parsers have been tried, and the consortium has agreed on the fact that the TableGen parser is very powerful. It parsed without any problem several legacy systems, among others, an embedded system in C++ of more than 1 million lines of code. Therefore, we have integrated TableGen into Audit-RE. Currently, TableGen parses only C++ programs. The TableGen parser saves relevant data about the sources in tables, and as a FAMIX/CDIF file. Information is stored about classes, methods, variables (attributes and others), accesses of variables, and method invocations, i.e. about all the entities of the FAMIX core model (see section 14.1). The tables can later be queried by programs written in the CQL-language, a dialect of the SQL query language. The FAMIX/CDIF file may be used to transfer information to other tools.

15.2 Automatic detection of violations of 'best-practice' heuristics

The best problem detection heuristics of the tool prototype Goose (see chapter 16) are implemented. This means that it is automatically checked whether the legacy program violates the principles stated in 'best practice' rules of a software design. The following rules are supported:

Remove unused components of a class

Base classes should not know anything about their derived classes

Avoid inheritance to achieve code reuse

Avoid multiple inheritance

Do not turn an operation into a class.

15.3 Object-Oriented Metrics

While working on the case studies many existing object-oriented metrics have been evaluated. The ones which seemed the most promising with respect to detecting problems in legacy code have been implemented in Audit-RE. These are:

Lines of Code (*LOC* (p. 22)): Measures the complexity of a piece of source code by counting the lines.

Depth in Inheritance Tree (*DIT* (p. 26)): Measures the depth of a class in the system's inheritance tree.

Number of Children (*NOC* (p. 26)): Counts the number of children (direct subclasses) of a class.

Number of Methods (*NOM* (p. 24)): Counts the number of methods in a class.

Number of Descendants (*NOD* (p. 26)): Counts the number of descendants (direct and indirect subclasses) of a class.

Response Set for a Class (*RSC*): Measures complexity and coupling properties of a class by evaluating the size of the response set of the class, i.e. how many methods (local to the class and methods from other classes) can be potentially invoked by invoking methods from the class.

Tight Class Cohesion (*TCC* (p. 25)): Measures the cohesion of a class as the relative number of directly connected methods. (Methods are considered to be connected when they use common instance variables.)

Change Dependency Between Classes (*CDBC* (p. 276)): Determines the potential amount of follow-up work to be done in a client class when the server class is being modified, by counting the number of methods in the client class that might need to be changed because of a change in the server class.

Data Abstraction Coupling (*DAC* (p. 24)): Measures coupling between classes as given by the declaration of complex attributes, i.e. attributes that have another class of the system as a type.

Weighted Method Count (*WMC* (p. 23)): Measures the complexity of a class by adding up the complexities of the methods defined in the class. The complexity measurement of a method is the McCabe cyclomatic complexity.

Reuse of Ancestors (*RA*): Measures how much of a class is reused from one of its superclasses.

15.4 Different views on the source code

Audit-RE provides several different, but synchronised views of the source code. The Application View supplies the list of files being parsed, the Module View displays the source code, the Query View shows the results of queries about the source code, and the Graphical View shows the different relations that exist between parts of the source code in a graphical way. All these views are synchronised: the selection of an item in a view will immediately update the corresponding parts in the other views. These views provide a good basis for browsing the code, and understanding the relations between the different objects, and the outcome of certain metrics and heuristics.

15.5 Example of the use of Audit-RE

The figure 15.1 shows a screen shot of Audit-RE. In the lower left hand side corner, we see the Query view. We have selected the metric 'Data Abstraction Coupling (DAC)'. This metric measures the coupling of a certain class with other classes. The central control classes of an application will normally have a

(relatively) high coupling with other classes. It is however undesirable to have classes with a high coupling which are not central classes. To detect the central control classes of an application, the Weighted Method Count (WMC) metric may be used as well. This metric measures the sum of the complexity of each method of a class. Classes with the highest WMC are usually the central classes of an application.

In the case studies we observed that all classes which have a high value for DAC, but which not have a high value for WMC, are candidate classes for improvement. These candidates are not the central control classes of the program, but they do have many dependencies with other classes. In the upper right hand side corner we see all these candidate classes with their parents and children, and the inheritance relations among them. If we click on a node in this filtered inheritance graph, we can inspect the implementation of that class in the Module view. This is shown in the lower right hand side corner of the picture.

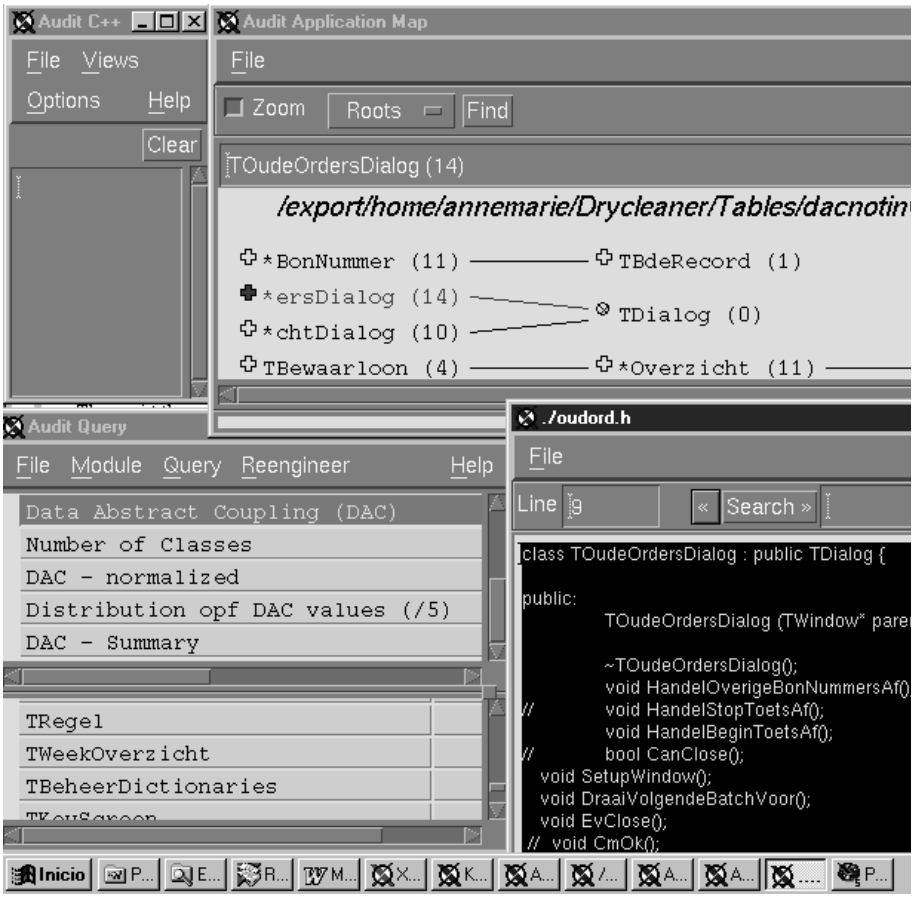


Figure 15.1: A screen shot of Audit-RE

15.6 Audit-RE and the reverse engineering and re-engineering patterns

How Audit-RE support the reverse engineering and re-engineering patterns mentioned in this book is explained in the following:

READ ALL THE CODE IN ONE HOUR **(p. 111)** The objective of this pattern is to make an initial evaluation of the software by walking through its code in a limited amount of time. Audit-RE supports this pattern by offering a good source code browser, integrated with an inheritance hierarchy browser.

GUESS OBJECTS **(p. 123)** The objective of this pattern is to progressively refine a model of a software system. Audit-RE supports this pattern by a metric called Weighted Method Count (WMC). During the various case studies of FAMOOS, we observed the fact that the outliers of this metric are the central, main control, classes of the project. They therefore offer a good starting point to progressively discover the model of the software system.

INSPECT THE LARGEST **(p. 131)** The objective of this pattern is to identify important functionality by looking at large constructs. Audit- RE supports this pattern by the metric 'NOM' (Number of Methods). This gives all classes ordered with respect to their number of methods, and by clicking on the classes with a high NOM, it is possible to view their implementation. Other counting metrics such as Number of Lines are also available.

VISUALIZE THE STRUCTURE **(p. 138)** The objective of the pattern is to obtain an overview of the software system's structure by means of well- known visualisations. In Audit-RE, several visualisations are supported.

TYPE CHECK ELIMINATION IN CLIENTS **(p. 177)** To detect single provider classes being used to implement what are conceptually a set of related types, we discovered during a case study that a good starting point will be the analysis of classes that gather much complexity in a few methods. These are classes with a high ratio of WMC (Weighted Method Count) and NOM (Number of Methods). This metric is provided by Audit-RE.

DISTRIBUTE RESPONSIBILITIES **(p. 203)** The objective of this pattern is to distribute the responsibilities equally among the classes of an object-oriented system to prevent large classes, which are hard to maintain and to reuse. Classes which have too many responsibilities can often be detected by applying the following guidelines (and their corresponding metrics):

- The largest classes in the system. As explained before (tool support for GUESS OBJECTS), these can be discovered by the Weighted Method Count metric, implemented in Audit-RE.
- Classes that use many other classes. These may be found by applying the metric 'Data abstraction Coupling', because this metric counts the coupling between a class A and other classes.
- The classes with a low cohesion, because these are classes which are used for different purposes. These may be found by applying the metric 'Tight Class Cohesion'.

Chapter 16

The GOOSE Tool-Set

Author(s): Oliver Ciupke and Markus Bauer

The ability to reengineer object oriented legacy systems and transform them into more flexible and extensible systems has become a matter of vital interest in today's software industry. Most reengineering projects however, are carried through in an unstructured and unfocused way, because methodology and tool support for such projects is poor.

During the FAMOOS project we have therefore developed methodologies and tool prototypes to show how this situation can be improved. This document describes one of these prototypes: the GOOSE tool-set.

16.1 The Problem

The size and complexity of modern object-oriented software systems makes reengineering tasks very difficult and costly: Before we can start any reengineering work on such systems, we have to understand their basic structure, how the different parts of the system work together, and what parts of the system should be reorganised and improved. However, for today's large systems we cannot gain that knowledge by just browsing through the system's source code. Instead, we need sophisticated tools that ease the task of getting a basic understanding of the system and that help us to focus our reengineering activities to the important parts of a system.

16.2 Principles and Tools

GOOSE is an experimental tool-set which was build to explore what kind of tool support is helpful for reengineering large C++ software systems. It focuses on the *model capture* and *problem detection* phases (see section 1.3) in a reengineering project. In particular, it features the following reengineering techniques:

- visualisation and grouping of software structures
- dependency analyses between different parts of the system
- automated checking of design heuristics
- metrics calculations

A reengineering session with GOOSE usually consists of several steps.

1. *Analyse the system's source code and build up a design database.* Before working with GOOSE, a design database is build up by analysing the system's source code. This design database holds information about all kinds of source code entities typically found in object-oriented systems and relationships among them. For example, it stores *subsystem, class, method* and *attribute* entities, and relations like *inheritsFrom, hasMethod, hasAttribute, callsMethod, acessesVariable,*.... Basically, this database contains the same information as the FAMIX-model, see 14.1.

   Although GOOSE supports different source code analysers (frontends), the *tablegen* analyser, which is shipped with GOOSE, is preferred for C++ systems.

2. *Visualise the design information.* After the database has been built up using *tablegen*, we can use it to visualise the structure of the system in manifold ways by converting its information into graph formats. These graph representations of the system can then be viewed with various graph editors and layout tools. GOOSE relies on the freely available third party graph visualiser *Graphlet* for this purpose.

   Graphical representations of a legacy system have proved very useful to get a basic understanding of the system and to find out, how different parts of the system work together. Sections 16.3.1, 16.3.2 and 16.3.3 provide some examples on how to use visualisations of the system for reengineering purposes.

3. *Query the database using* PROLOG *in order to check design guidelines.* GOOSE's design database contains the most important design information about a system. By converting this information into PROLOG facts, we can evaluate the design of the system using PROLOG queries. In section 16.3.4 we describe, how this feature of GOOSE can be used to check violations of certain design guidelines.

4. *Apply abstractions to the database.* Since most object-oriented systems are very large, abstractions should be applied to the database. GOOSE provides the *reView* tool for that purpose. Using *reView*, we can filter out unrelevant parts of the database and and group source code entities into more abstract entities in order to get a clearer view on the system's structure (see section 2.3). *reView* also features some basic metric calculations (see section 2.1) for an introduction on how to use metrics during reengineering projects). Even more metrics can be applied by using the *Audit-RE* environment (see section 15). For that purpose, GOOSE provides an interface to *Audit-RE*.

Figure 16.1 gives an overview on these functionalities and how they relate together.

16.3 Usage Scenarios

In this section, we describe in some scenarios how we can analyse a legacy system using GOOSE. These scenarios are based upon our experiences in various case studies during the FAMOOS project.

16.3.1 Get an Overview on the System's Architecture

When starting work on a reengineering project, we often want to get an overview on the system's structure. (Frequently reengineering projects are done by software engineers who where not involved in the initial development of the system, therefore they do not have much knowledge about its structure and its inner workings.)

To get an overview over the legacy system, we use *Graphlet* to visualise GOOSE's design database. Figure 16.2 shows such a visualisation of a large C++ system[1]. In that visualisation, we have used GOOSE to

[1] All examples in the next few sections come from an industrial case study. Due to non-disclosure agreements we have changed the names of the system's subsystems and classes. Apart from this, we have not changed anything else in these examples. Therefore, they represent authentic reengineering situations.

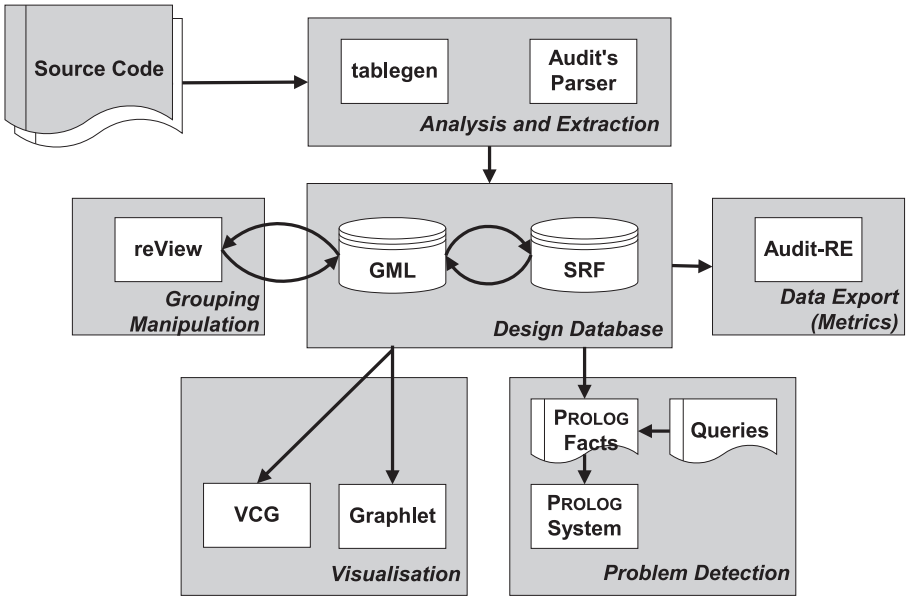


Figure 16.1: The GOOSE tool-set

simplify the database using filtering and grouping techniques. The entities of the system (classes and methods) have been collapsed *(grouped)* into subsystem nodes (as defined by the directory structure of the system's source code – each directory in the source code tree is considered to represent a subsystem). Thus, the visualisation shows the subsystem dependency graph of the system. A (directed) edge between two subsystem nodes in the graph indicates that classes (or methods) of one subsystem depend on (or use) classes (or methods) in the other subsystem.

From that visualisation of the system's architecture we can learn, that the system is implemented in four layers.

16.3.2 Check Dependencies

Usually, once we have found out what basic architecture the system has, we are interested if there are spots in the system where this architecture is not respected. To continue our example from section 16.3.1, we assume that the following two principles apply to our legacy system:

1. The system is organised in layers (we discovered that in section 16.3.1). Upper layers (denoted with higher numbers in figure 16.2) implement their functionality by using lower layers (lower numbers), however, lower layers *should not* access upper layers.
2. The system actually consists of two parts: The first part is a framework (which might be used in other systems as well), the second part contains product specific code, i.e. code that instantiates the framework and implements a product by using the framework. It should be clear, that framework code *should not* depend on product specific parts. (In figure 16.2, framework parts are denoted with the suffix *FW*, product specific parts have the suffix *Product*.)

With GOOSE, we can search violations to these two principles. Figure 16.3 shows a visualisation of this: we have filtered out all dependencies except for those violating the principles. As we can see, there are some cases, where lower layers access upper layers. For example, *Layer1/Product* acess *Layer3/Product*. Additionally, there are some framework parts depending on product specific parts. In lowest layer of the system, layer 1, for example, *Layer1/FW* depends on *Layer1/Product*.

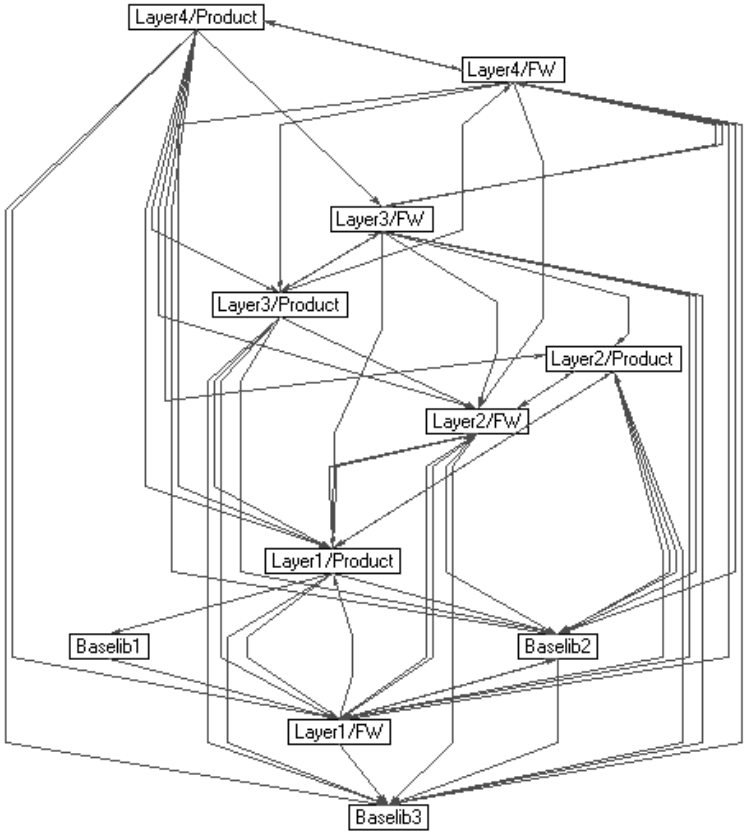


Figure 16.2: Visualisation of the system's architecture

To investigate this issue a little bit further, we can use a more detailed visualisation. Figure 16.4 shows the *Layer1/FW* and the *Layer1/Product* subsystems at a class level: nodes represent classes of the two subsystems (product specific classes are dark, framework classes have white colour), edges either mean that a class belongs to a subsystem (radial edges, leaving from the *Layer1/FW*, *Layer1/Product* nodes), or inheritance relationships. A closer look at the inheritance edges reveals, that some framework classes inherit from product specific classes.

This may cause a lot of problems, since such a dependency is counter-intuitive to most design styles. Now, that GOOSE helped us to find these spots, we can look into the source code of the affected classes and maybe improve the code in order to make it more understandable and conform to the architecture's principles.

16.3.3 Verify a Subsystem Structure

Often, we are interested to know how well a subsystem structure is thought out. Usually, a subsystem structure can be considered as well designed if classes *inside* a subsystem are significantly related (i.e. there are some dependencies between them), and if there are not to many dependencies between classes in *different* subsystems.

To verify, whether a subsystem adheres to this rule, we can use visualisations. Figure 16.5 shows a subsystem structure consisting of three subsystems.

Again, nodes represent classes, edges represent relationships between classes. Classes are colored according to the subsystem they belong to: white, gray and dark. The graph has been layouted using a *spring*

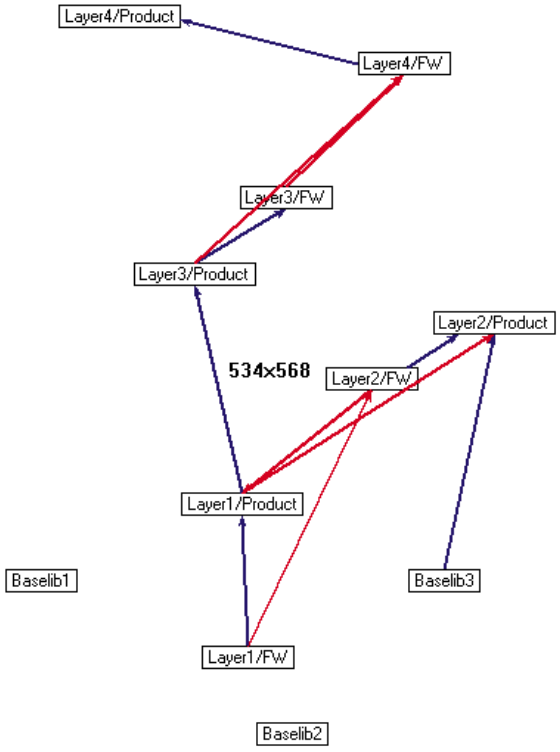


Figure 16.3: Violations of the system's architectural principles

layout. This means, that nodes are grouped together as if the edges where springs (or rubber bands), pulling the neighbour nodes of a node back to the node against some natural force that tries to keep them away. As we can see, the layout algorithm nicely groups the classes of each subsystem together. We note, that this is mainly due to the radial subsystem containment edges, which denote which classes belong to which subsystem. (Like in section 16.3.1, the subsystem structure has been derived from the directory structure of the system's source code.)

We now remove those artificial subsystem containment edges (remember, classes have been placed into the subsystems, or source directories, by the developers and we want to verify if they were correct in doing so) and apply the spring layout algorithm again to that structure. Figure 16.6 shows the result. We can still see that the classes are again mostly grouped according to their subsystems – very dark classes are more at the bottom part of the graph, most of the white classes can be found somewhere in the middle, and the gray shaded classes are in the upper part of the figure. However, we can see, that especially some of the very dark and some of the white classes have "escaped" into other subsystems.

When reengineering the system, we should consider (after carefully looking at the source code) to move these classes into the subsystems they "escaped" to.

16.3.4 Search for Violations of "Good" OO Design

One of the most time consuming tasks during reengineering is finding out *where* the system has to be changed and *what kind* changes are needed in order to achieve a more flexible and extensible software structure. We can ease this task by searching for violations of certain design guidelines [RIEL 96].

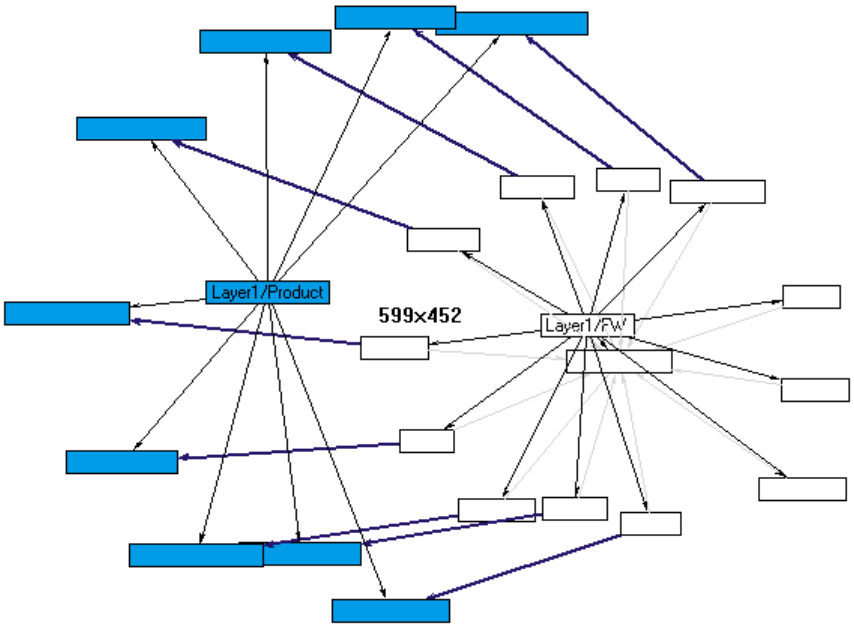


Figure 16.4: An architecture violation in more detail

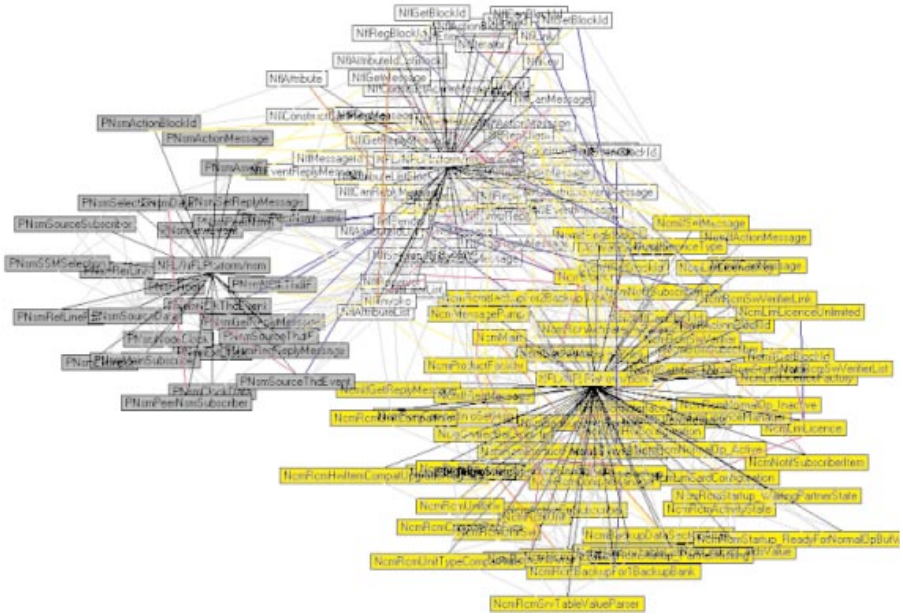


Figure 16.5: Verify a subsystem structure – step 1

Design guidelines (or heuristics) make a statement about good design – a violation of such a guideline may indicate a design problem. The following example shows a typical design guideline.

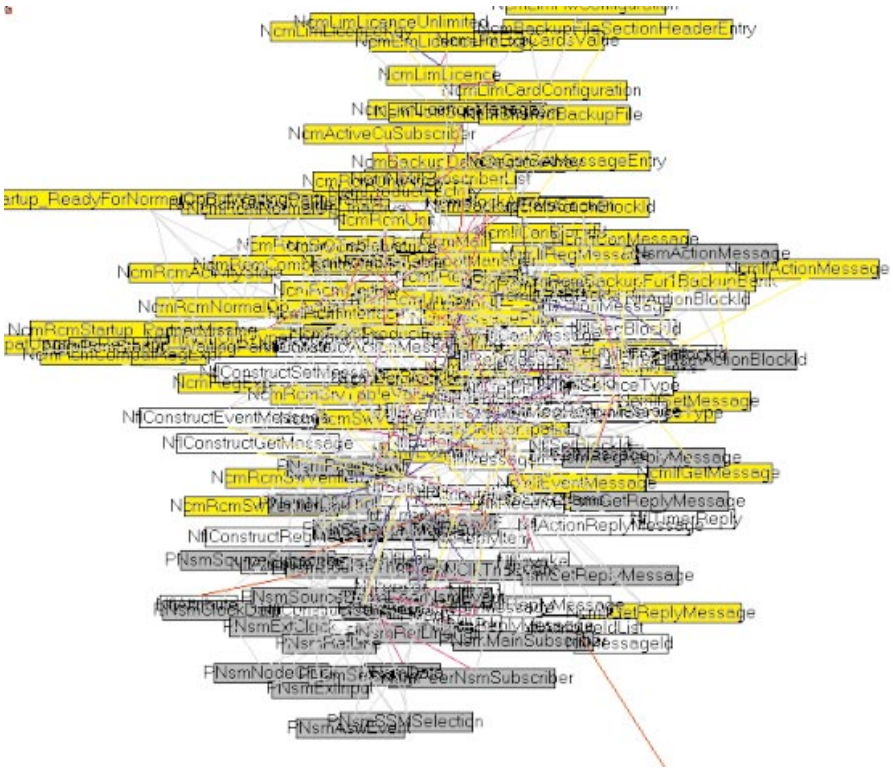


Figure 16.6: Verify a subsystem structure – step 2

> "Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes." [RIEL 96]

If base classes depend on their descendants, then these descendants cannot be altered independently of their ancestors. But exactly this is often needed during the evolution of an object-oriented system. Thus, a violation of this guideline points to a spot in a software system that is hard to change or maintain.

GOOSE be used to check this guideline automatically. To do this, the rule must be formalised as a PROLOG query, which can then be used against the PROLOG representation of GOOSE's design database. Figure 16.7 shows a possible PROLOG formalisation of the design guideline introduced above.

After running this formalised PROLOG query, GOOSE prints out all occurrences in the system that violate the design guideline. Additionally, offending classes can be highlighted in a visualisation view. To give an example, figure 16.8 displays a class structure from the publically available *ET++ application framework* (see [WEIN 89]), that violates our design guideline. The figure shows the offending classes as nodes in the graph. Dark arrows represent inheritance relationships between these classes, light arrows the existence of method calls between methods of these classes. Class `Clipper` inherits from `VObject` directly and from `EvtHandler` transitively. Both `EvtHandler` and `VObject` contain calls to a method of `Clipper`. This means that changes to `Clipper` may require `EvtHandler` and `VObject` to be changed as well. Both can be found at a high position in the inheritance hierarchy, where changes affect many further classes in turn: Figure 16.9 illustrates, that a lot of classes (more than 370) depend on `EvtHandler` and `VObject`. We can conclude that this structure is problematic, since a single change in a subclass results in unexpected changes in some of its superclasses, and these changes, in turn, affect almost the whole system.

A possible reorganisation which could simplify this situation would be to introduce abstract base-classes or migrating methods or attributes between classes. Which of the solutions are suitable depends on the overall structure and on the particular goals and requirements when reengineering a specific system.

```
% Base classes should not have knowledge about their descendants
knowsOfDerived (Class, DerivedClass) :-
    % Both Class and DerivedClass must be classes
    class (Class), class (DerivedClass),
    % DerivedClass is a direct or transitive descendant of Class
    trans (inheritsFrom, DerivedClass, Class),
    % The base class knows its heir
    knows (Class, DerivedClass).

% A class 'knows' another class, if
knows (Class1, Class2) :-
    % it inherits from that class, or
    class (Class1), class (Class2),
      inheritsFrom (Class1, Class2);
    % it has an attribute of that type, or
    hasAttribute (Class1, Attr), hasType(Attr, Class2);
    % it has a method which returns an object of that type, or
    hasMethod (Class1, Meth1), returns (Meth1, Class2);
    % it has a method which calls a method of that class, or
    hasMethod (Class1, Meth1), calls (Meth1, Meth2),
      hasMethod (Class2, Meth2);
    % it has a method containing a parameter with type of that class.
    hasType(Param, Class2).
```

Figure 16.7: Formalisation of the design rule in Prolog

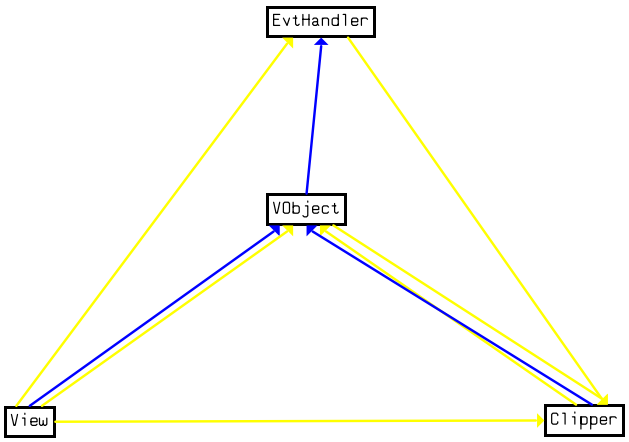


Figure 16.8: A class structure from ET++ violates the design guideline

To make the automated checking of design heuristics more practicable, GOOSE comes with a set of readily applicable, formalised guidelines. Our experiences have shown that most of these guidelines produce very helpful results. GOOSE often pointed us to real problems in legacy systems, and by fixing them, we were able to improve the system significantly.

16.3.5 Use Metrics to Quantify Results

There are manifold ways to use object-oriented software metrics during reengineering projects. To support this, GOOSE provides an interface to the *Audit-RE* environment. An introduction on how to use metrics in reengineering projects can be found in 2.1.

To continue the example from section 16.3.4, we use the complexity metric *WMC* (see page 23) and the coupling metric *DAC* (see page 24) to underpin the observation we made there: Changes to the class `Clipper` in *ET++* are considered harmful. As we have already noticed above, such changes may require changes in `VObject` as well. The *WMC* metric, however, shows that `VObject` is one of the complexest

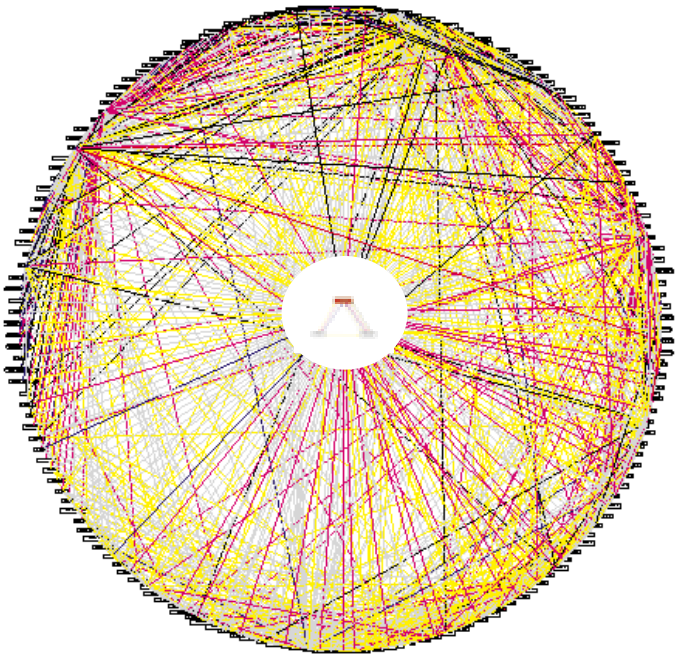


Figure 16.9: The detected problem affects lots of classes

	WMC	DAC
`VObject`	148	17
Avg. value (system-wide)	27.5	5.9
Max. value (system-wide)	223	29

Figure 16.10: *WMC*- and *DAC* measurements for *ET++*'s `VObject`

classes in *ET++*, and the *DAC* metric shows, that `VObject` is tightly coupled with the rest of the system. (Figure 16.10 contains the corresponding measurements, high *WMC*-values indicate a high complexity of a class, high *DAC*-values a tight coupling with the rest of the system.) This means that if we would like to change `Clipper`, we would probably have to understand `VObject`, which is difficult because of its complexity and its tight coupling with other classes. Knowing about this fact is important – we now have a feeling on how much effort we have to spend to safely apply changes to `Clipper`, or `Vobject` respectively.

16.4 Implementation Information

GOOSE was intended to be a prototype to evaluate different reengineering techniques and to investigate, how these engineering techniques can be supported by a tool. Therefore, GOOSE is implemented in a very extensible way: GOOSE consists of simple, command-line driven tools. Each tool implements one single, specific functionality. This way, each tool can easily be replaced by an improved version. The most important tools shipped with GOOSE are:

- *Tablegen*, to analyse the source code and build up the GOOSE design database.

- *Conversion scripts*, to convert the GOOSE design database into different formats. Such tools are needed, for example, to convert the GOOSE database to the *GML*-format, which is used by the visualisation tool *Graphlet*, or to convert it into PROLOG facts for automatic problem detection and checking design heuristics.
- *reView*, to manipulate the GOOSE database (grouping and filtering).

Most tools have been developed using the *Perl* scripting language, *tablegen* and *reView*, however, are implemented in C++.

To run, GOOSE requires two additional third party tools:

- *Graphlet*, to visualise the system. *Graphlet* is a graph editor and layout tool. It is freely available from the University of Passau[2].
- A PROLOG system. GOOSE uses *ECLiPSe*[3], but GOOSE can probably adapted to other PROLOG systems as well.

As of today, GOOSE requires the *SUN Solaris/SPARC* operating system to run.

Since GOOSE consists of single, command-line based tools, it requires some training on the user's part. Experiences with a Unix-like operating system are recommended. However, there are recently efforts to improve the usability of GOOSE. Most of the tools have been glued together by a makefile driven mechanism, which automates the most common tasks that ahve to be executed during a reengineering session with GOOSE. Further improvements are expected, as GOOSE is being extended by a graphical user interface.

16.5 Contact Information

GOOSE has been implemented at the Forschungszentrum Informatik (FZI), Karlsruhe. The first version of GOOSE has been implemented by Holger Bär and Oliver Ciupke – various other authors have extended it and contributed new tools.

Further information and a current version of GOOSE can be obtained from Oliver Ciupke, `ciupke@fzi.de` or http://www.fzi.de/ prost.html.

[2] see http://www.uni-passau.de/ Graphlet
[3] see http://www.ie.utoronto.ca/ EIL/ ECLiPSe/ eclipse.html

Chapter 17

DUPLOC

Author: Matthias Rieger

Duplicated code is an ubiquitous phenomenon which entails copied errors, software that is hard to change and unnecessary code bloat. Detecting duplicated code is a reverse engineering task desperately in need of tool support since without *a priori* knowledge about the loci of duplication, detecting code duplication is tantamount to finding the needle in the haystack. DUPLOC is a tool which helps in the visualisation of code duplication and provides the data for statistical analysis of the duplication.

17.1 Problem

Code duplication happens all the time. Every developer now and then clones pieces of software. When encountering a familiar problem that one has (seen) solved before, it is a normal reflex to reuse the existing code. However, the time to abstract and generalise the existing solution so that one implementation fits the old *and* the new context, is often missing because of deadline pressure. Thus the code is copied, pasted and altered slightly until it fits.

Software that lacks the property of *"everything is said once and only once"* is generally harder to change, because for every piece of code that is changed, all the clones must probably be changed as well. Where this "just" brings more work to the maintainer, the problem of errors that are copied along with the code is more dangerous. Fixing the error in one place then does not mean that the error has been exterminated from the entire system until one has found and fixed all the clones.

17.2 Principle and Tool

Duplication Detection Since any part of source code can be copied, we cannot search for specific program clichés but rather have to deduce them from the input itself by comparing every line with every other line of the system. This comparison produces an enormous amount of data which can be represented in a two-dimensional matrix. Each cell of the matrix stands for a comparison between two lines of code and contains a value unless the two lines did not match [HELF 95]. The next step is then to find zones of interest in the matrix. The matrix can be examined using either an automatic or an exploratory approach:

Automatic Examination: A tool searches for some known configurations of dots in the matrix and delivers a report of the instances that were found. This approach finds interesting spots automatically which is convenient and efficient for large systems. The objects of the automated search can range from simple diagonals with holes like in the example Figure 17.1 *b)* to dot clusters found using statistical methods.

Visual Exploration: Using a tool that displays the comparison matrices graphically, the engineer browses through the matrix, zooms in on interesting spots, and, by clicking on the dots, examines the source code that belongs to a certain match. This exploratory approach and can lead to unexpected findings (see the examples in [HELF 95]).

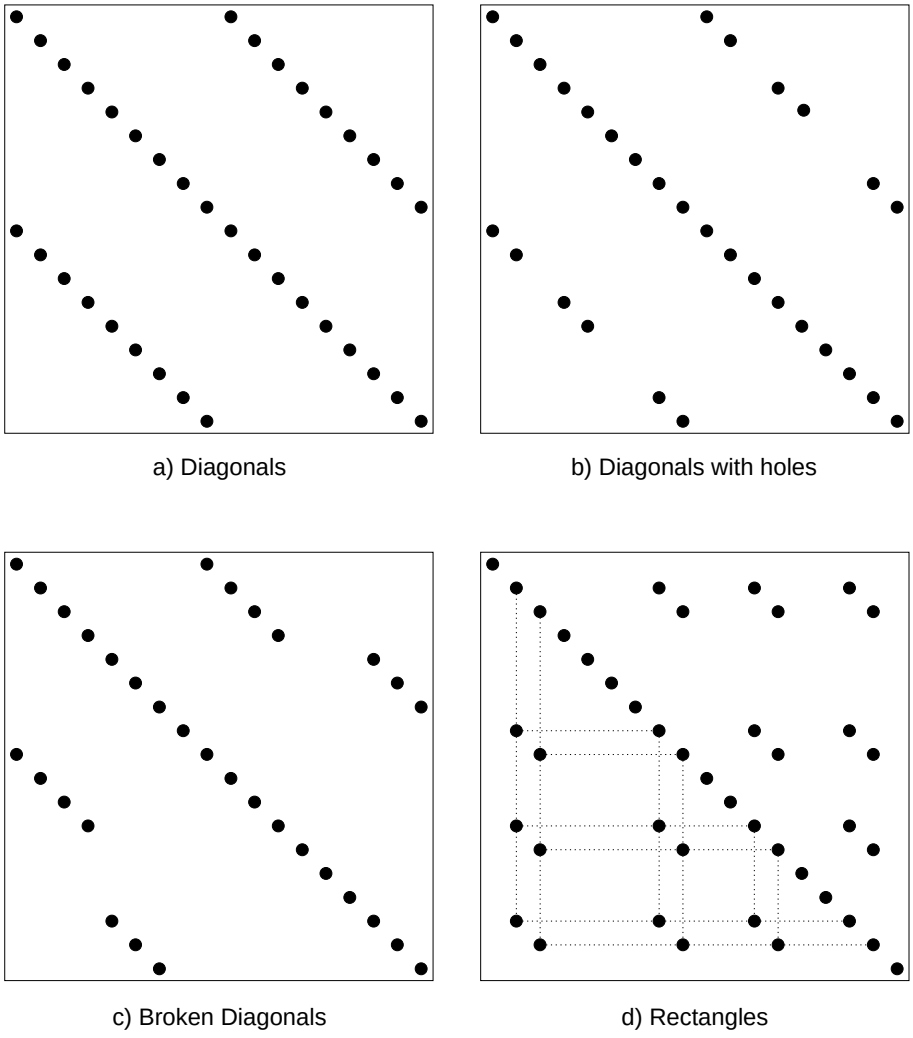


Figure 17.1: Different Configurations of Dots.

Some interesting configurations formed by the dots in the matrices are the following:

- diagonals of dots indicate copied sequences of source code (see Figure 17.1 a)).
- sequences that have holes in them indicate that a portion of a copied sequences has been changed (see Figure 17.1 b)).
- broken sequences with parts shifted to the right or left indicate that a new portion of code has been inserted or deleted (see Figure 17.1 c)).
- rectangular configurations indicate periodic occurrences of the same code. An example is the `break` at the end of the individual `cases` in a C/C++ `switch` statement or recurring preprocessor commands like `#ifdef SOME_CONDITION` (see Figure 17.1 d)).

Language Dependence Code duplication detection tools in general transform source code into some intermediate format upon which the comparison algorithm works. These code transformations usually seek to hide code elements which were most likely changed during the copy&paste editing. For example, to catch duplicated code where identifiers have been renamed for the new context, the code transformation replaces identifiers with a generic name, e.g. id.

The formats range from tokenized texts [BAKE 95] to abstract syntax trees [BAXT 98] to tuples of metric values [MAYR 96b]. All these sophisticated transformations require some form of parsing or at least lexical scanning of the source code. Since in practice it can be difficult to find a parser for the language—or even for the language *dialect*—at hand [BAXT 98], the application of a parser-based approach is sometimes hindered.

17.3 The tool

The DUPLOC tool compares source code line–by–line and displays a two-dimensional comparison matrix.

17.3.1 Features

DUPLOC reads source code lines and, by removing comments and white space, generates a "normal form"-representation of a line. These lines are then compared using a simple string matching algorithm. DUPLOC offers a clickable matrix which allows the user to look at the source code that produced the match as shown in Figure 17.2. DUPLOC offers the possibility to remove noise from the matrix by 'deleting' lines that do not seem interesting, e.g. the public: or private: specifiers in C++ class declarations. Moreover, DUPLOC offers a batch mode which permits to search for duplicated code off line. A report file, called a *map*, is then generated which lists all the occurrences of duplicated code in the system.

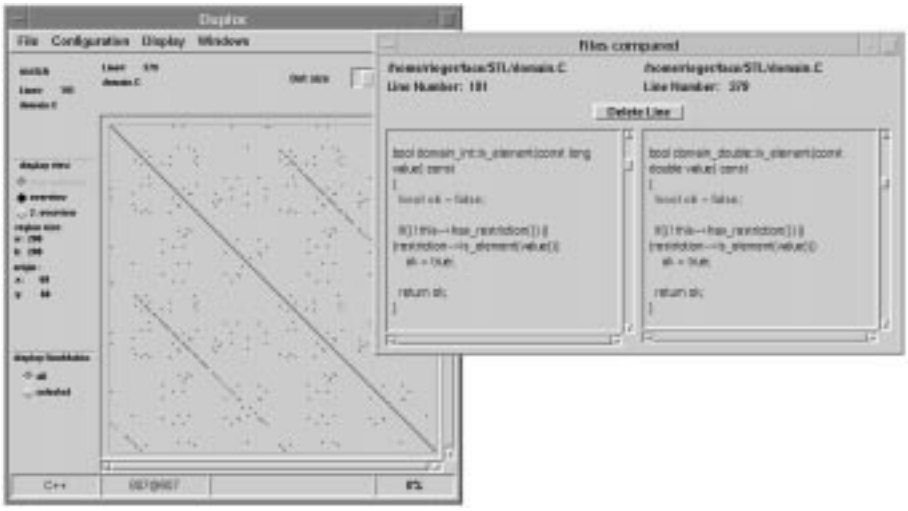


Figure 17.2: The DUPLOC main window and a source code viewer

17.3.2 Evaluation

The tool evaluates favourably according to the following requirements which were formulated for re-engineering tools in [HAIN 96], [LANG 95]:

- *Exploratory.* The visual presentation of the duplicated code allows the user to detect structures or configurations of dots that are unknown or new. The tool does not impose a certain view on the data, making unexpected findings possible.
- *Browsing views.* The tool allows one to look at the whole system or at specific parts (files) of it. Scrolling and zooming support is provided.
- *Language independent.* Due to the simplicity of the comparison algorithm (string matching), we do not depend on parsers for a specific language.
- *Forgiving.* The comparison is not based on source text that has been parsed, so there can be any number of syntax errors in it which do not break the algorithm.
- *Complete.* The *map* gives us an overview of the structure of code duplication that occurs in a system as a whole.
- *Configurable.* The comparison algorithm is easily exchanged. The matrix display can be enhanced giving different grey– or colour–values to the individual dots to show the percentage of a fuzzy match. It is then also possible to visualise another data dimension. This could be used, for example, to identify different kinds of statements.

The drawbacks of the approach can be summarised as follows:

- *Detection Only.* The tool can help in detecting occurrences of duplicated code. It does not help to actually decide what should be done with the code, e.g. if it should be left untouched or be refactored. The automatisation of this step will always be restricted to very few, unambiguous cases.
- *Syntactic level.* The tool compares only syntactic elements of the program. This implies that it does not discover duplicated *functionality*.
- *Comparison Algorithm.* Using a simple comparison mechanism like string matching, code parts that were slightly changed will not be recognised. Other algorithms can compute fuzzy match values.

17.3.3 Outlook

Some features that DUPLOC does not yet have, but are desirable for code duplication detection, are the following:

- detection of parameterised matches, e.g. duplications where names of variables have been systematically altered. Duplicated code with this property is a candidate for refactoring into a function. Parsing facilities for the language supported must be available.
- interpretation of dot configurations to detect meaningful ones. Ideally, a specific dot configuration should be interpretable as a hint at what kind of reengineering procedure should be applied.

17.4 Implementation Information

DUPLOC is implemented in SMALLTALK and runs on version 2.5 and 3.0 of VISUALWORKS . This makes the tool platform independent (it runs on UNIX, Mac and Windows platforms). DUPLOC is integrated with the the VISUALWORKS framework and the ENVY development environment, making it possible to directly compare SMALLTALK classes or ENVY applications.

17.5 Contact information

DUPLOC is being developed by Matthias Rieger as part of a PhD work in the Software Composition Group at the University of Berne. DUPLOC is freely available under the GNU Public License at http::/www.iam.unibe.ch/~scg/ Archive/ Software/ Duploc . The author can be contacted at rieger@iam.unibe.ch .

Chapter 18

CodeCrawler

Author(s): Michele Lanza and Stéphane Ducasse

CODECRAWLER is a tool that supports reverse engineering of large object-oriented projects. It combines the immediate appeal of visualisations with the scalability of metrics. Furthermore, it allows the user to tailor what information is presented as well as how it is presented.

18.1 Problem

The reverse engineering of large object-oriented legacy systems benefits greatly from an approach providing a fast overview and focusing on the problematic parts. Among the various approaches that exist today, two are particularly interesting for large scale reverse engineering. One is *program visualisation*, often applied because good visual displays allow the human brain to study multiple aspects of complex problems in parallel[1] [CONS 92], [KLEY 88], [LAMP 95], [MÜ 86], [PAUW 93], [JERD 97], [SAND 96], [STOR 95], [SUGI 81], [CROS 98], [BALL 96], [JERD 97]. Another is *metrics*, because metrics are known to scale up well [DEME 99a], [KONT 97], [LEWE 98a],[LORE 94],[MARI 98].

We propose an tool encompassing both graph visualisation and metrics combined in a simple approach where (a) the graph layout is very simple and (b) the extracted metrics are straight forward to compute. Indeed, our goal is to identify useful combinations of graphs and metrics that can be easily reproduceable by reverse engineers using a scriptable reengineering tool-set.

18.2 Principle and Tool

Combining Graph and Metrics. We enrich a simple graph like tree, boxes one besides the other, with metric information of the object-oriented entities it represents. In a two-dimensional graph we render up to five metrics on a single node at the same time. For a more in-depth discussion of this topic please read Section 2.2.4.

1. **Node Size**. The width and height of a node can render two measurements. This means that the wider and the higher the node, the bigger the measurements its size is reflecting.
2. **Node Position**. The X and Y coordinates of the position of the node reflect two metric measurements. This requires the presence of an absolute origin within a fixed coordinate system. Note that not all layouts exploit this dimension.

[1] This is often phrased as "One picture conveys a thousand words".

3. **Node Colour**. We use the colour interval between white and black to display yet another measurement. The higher the value the *darker* the node is. Thus light gray represents a *smaller* metric measurement than dark gray.

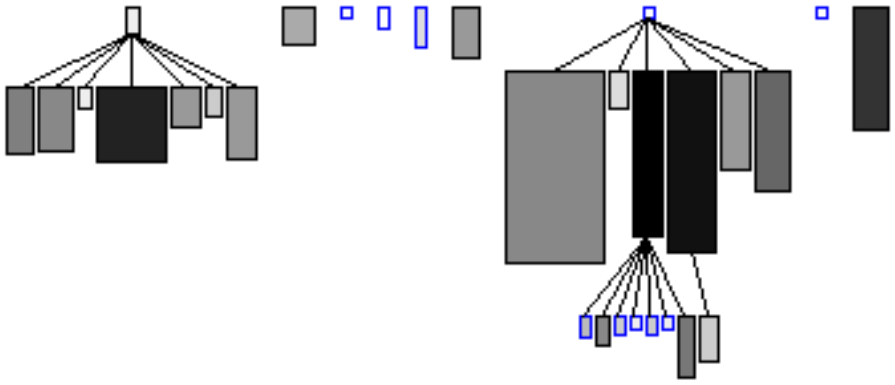


Figure 18.1: Inheritance Tree; node width = NIV, node height = NOM and colour = NCV.

Illustration. As an example Figure 18.1 shows an inheritance tree graph from CodeCrawler. The nodes represent the classes and the edges represent the inheritance relationships. The size of the nodes reflects the number of instance variables (width) and the number of methods (height) of the class, while the colour tone represent the number of class variables. In this case the position of the nodes does not reflect metrics as the position is defined by the layout algorithm.

18.2.1 The Tool.

CODECRAWLER is an open platform providing a graphical representation of source code combined with object oriented metrics. Besides the pure combination of graph with metrics values, while building CODECRAWLER we were confronted with practical considerations such as the minimal size of a node or the size of the screen.

Graphical Considerations and Influences. For the node size, we chose to implement the mapping such as to really reflect the measurement in the size on the screen with a slight distortion in the case where the measurement drops below a certain threshold. A minimal node size is a purely practical issue that is necessary when we want the graph to be interactive, since clicking with the mouse pointer on nodes only 1 or 2 pixels wide is unecessarily difficult.

While the usage of different colours is a good way to attract the attention of the eye, the usage of too many colours should be avoided. It results in an optical overload that hinders rather than helps understanding. The solution with the colour tone has the advantage that numerical information can be transmitted by colours: we map numerical values (e.g. the metric measurements) into a colour interval ranging from white to black. Although this is a good way to display a supplemental metric we must notice that since the perception of a colour tone is less precise than the perception of size, the colour metric is only useful for the detection of extreme values.

Note that CODECRAWLER supports different distributions (linear, logarithmic...) to represent the size of the nodes plus different modes like the shrinking of the graphs to fit the graphs into the size of a screen. It is also able to mark nodes whose metrics exceed a certain threshold value.

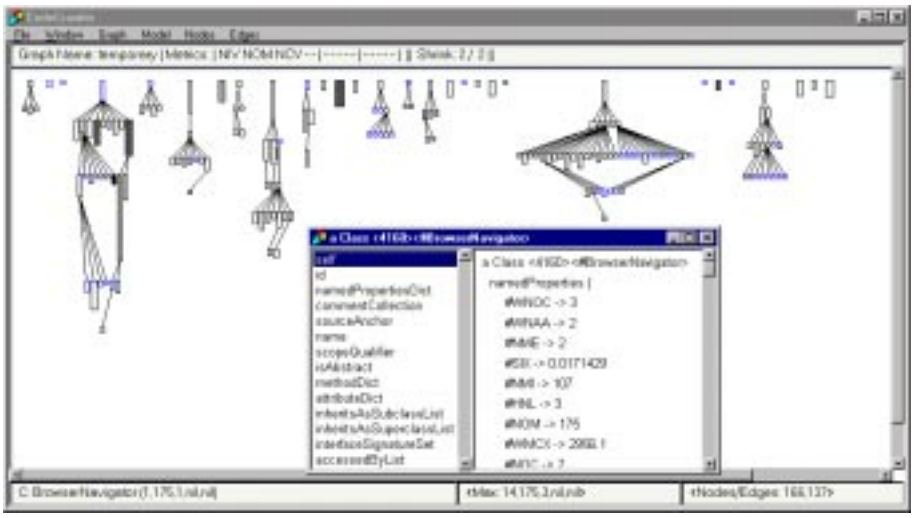


Figure 18.2: The CodeCrawler platform at work: (1) an inheritance tree with x node size = NIV, y node size = NOM, colour = NCV and (2) inspecting the data of a represented entity.

Supporting Reverse Engineering. Besides the definition of concrete graphs and operations on the graphs themselves like highlighting all the edges arriving at a specific node, following a particular edge, applying a new graph to a specific node . . . , CODECRAWLER provides a number of other features that greatly enhance reverse engineering activities. Most noteworthy are (1) the query of the graph to identify a node according to some criteria such as its name and (2) code navigation via the graph. Each graph entity is linked to the code entity that it represents, so the reverse engineer can browse the code related to the displayed entity as well as its metrics.

Moreover, as shown in Figure 18.2 CODECRAWLER displays the information of the current displayed graph (top border) and the information related to the entity under which is the mouse (bottom border). In Figure 18.2 the metrics are NIV, NOM and NCV applied on class entities, the last investigated class is BrowserNavigator that has 1 instance variable, 175 methods and 1 class variable.

18.3 A Scenario

The scenario itself consists of various kinds of graphs, some of them providing overviews of the classes and methods in the system, others focusing on possible problems in the design.

The scenario conveys well how customisable and exploratory a hybrid approach can be. Indeed, the idea is that different graphs provides different yet complementary perspectives. Consequently, a concrete reverse engineering strategy is to apply the graphs in some order, although the exact order may vary depending on the kind of system at hand and the kind of questions driving the reverse engineering project.

The particular software system used for our scenario is the Refactoring Browser [ROBE 97a] which is well-known throughout the Smalltalk community. To give an idea about the size of the system: the Refactoring Browser consists of 166 classes (not counting the metaclasses), 2365 methods, 365 instance variables, 2198 instance variable accesses and 9780 method invocations. In that respect we consider it a small to medium case study.

Note that we have run other experiments on industrial case studies implemented with C++ and SMALLTALK . Unfortunately, due to non-disclosure agreements with the case study providers, we cannot publish the results of these studies here.

18.3.1 Understanding the Refactoring Browser

1. Class Size Overview: Checker graph. One of the first impressions of the system that reverse engineers desire is a feeling for the raw physical measures of a system. For that purpose, we generate a so called *checker graph* with lines of code as node size, the number of instance variables as colour tone and we sort the nodes using lines of code as criterion (see Figure 18.3).

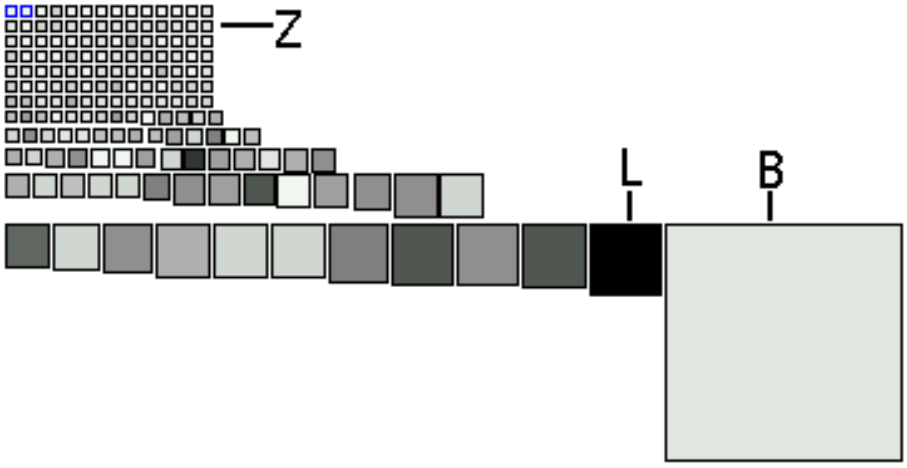


Figure 18.3: Class Size Overview via Checker Graph; node size = LOC, and colour = NIV.

Interpretation. The checker graph is useful for showing relative proportions between the system elements, in this particular case it shows the proportion among the classes of the software system in terms of lines of code. Through sorting it is easy to identify the largest and smallest classes. In this graph the biggest node represents the class BrowserNavigator with 1495 lines of code identified as B. The second biggest class with 441 lines of code is called BRScanner identified as L in Figure 18.3. We are also able to see that many classes in the system (marked as Z) are very small, and that there are some empty classes which are positioned on the upper left corner of Figure 18.3. (This last detail is only visible on the screen and not on the paper version, because metric measurements equal to zero make the nodes to be displayed with a blue border and this colour is not rendered in this outprint).

2. Inheritance Overview: Trees. To assess the size and complexity of the system, we request for an *inheritance tree*. We use as node size the number of instance variables and the number of methods, while the colour tone represents the lines of code of the classes (see Figure 18.4).

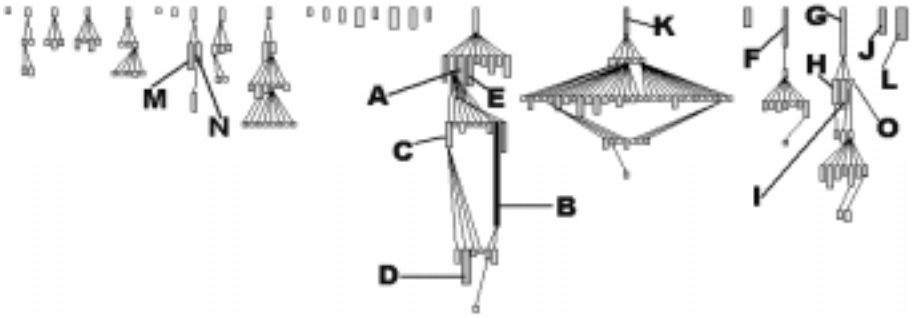


Figure 18.4: Inheritance Overview via Tree; node width = NIV, node height = NOM and colour = WLOC.

Interpretation. We observe a few main hierarchies with a high proportion of very small classes. Then, using the number of methods per class as a criterion we identify some candidate classes for further investigation: (1) the smallest class (O) is completely empty and (2) 12 classes have from 40 methods per class to 175 at the maximum (B).

These 12 classes can be classified according to their position in the inheritance tree: being a leaf (D, I), being on top of a hierarchy (F,G,K), being in the middle of the hierarchy (A, B) or being alone (E,J,L). Sibling classes like (H, I) and (N, M) are good candidates for refactoring analysis to see if some of the code could not be refactored up in their superclass.

An example of possible further investigation is the huge class called BrowserNavigator that implements 175 methods (named B in Figure 18.4) whereas its superclass Navigator (A) already implements 70 methods. Another interesting class without subclasses is the class called BRScanner (named L), that implements 49 methods and defines 14 instance variables.

3. Focus on Class Cohesion: Confrontation Graph. Given the analysis of Figure 18.4, we will focus on the class BRScanner (L). More precisely we want to understand the internal coupling of the class by looking at the way the methods access its instance variables. Therefore, we apply a *confrontation graph*: a graph where an edge between an instance variable and a method represents an instance variable access done by the method. The resulting graph is shown in Figure 18.5. The instance variables are the middle row of nodes and the methods are the top and bottom row of nodes.

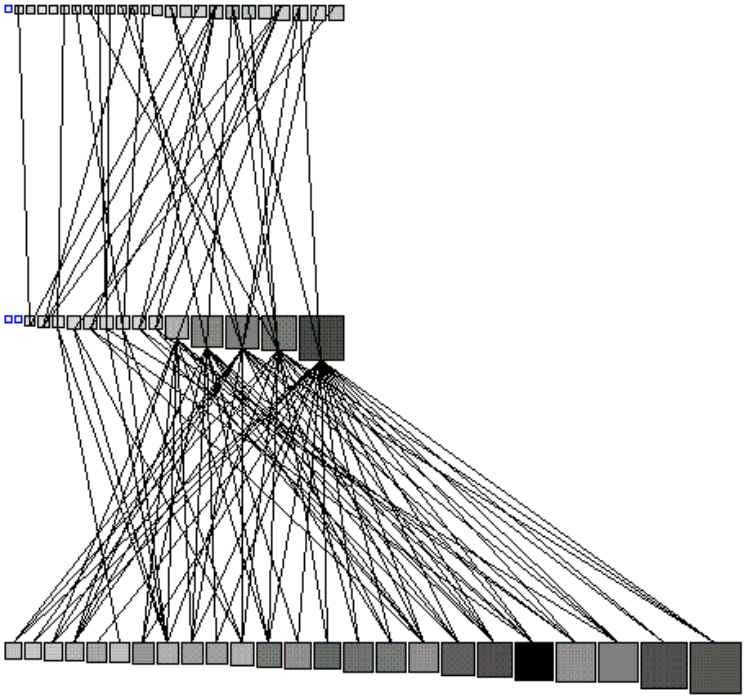


Figure 18.5: Focus on Class Cohesion of BRScanner via a Confrontation Graph; method node height and width = NOS and colour = LOC; attribute node height and width and colour = NAA.

Interpretation. The confrontation graph reveals that there are no apparent clusters in the way methods access instance variables. This is a sign that this class is quite cohesive making a split difficult if not impossible. However, it also implies that subclassing will be quite difficult. Note that in other experiments, we have —by means of the same confrontation graph— discovered classes that can easily be split. Unfor-

tunately for our scenario, but good for the Refactoring Browser, we did not find such case in the case study described here.

18.3.2 Scenario evaluation.

Results. Using a hybrid reverse engineering approach combining metrics and graph layouts to reverse engineer the Refactoring Browser provides us with a quick understanding of the system without having to dive into the details of the system. Due to space limitation we could not show system of all graph layouts and metrics, nor could we include more useful graphs. Interested readers may consult [LANZ 99] for a more complete description.

- **Fast overview**. The *checker graph* provided us with a intuition for the proportions in terms of code size and helped us to quickly identify extreme cases. The inheritance overviews helped us to identify and qualify the main hierarchies: the graphical navigation and tool classes like BrowserNavigator, the refactoring classes, the abstract syntax tree representation, the parser and the scanner. More than just displaying hierarchies, CODECRAWLER helped us to understand the quality of the hierarchies: for example the refactoring hierarchy whose root is K in Figure 18.4 is composed by a high number of small classes whereas the abstract syntax tree hierarchy, whose root is G in Figure 18.4, is composed by more substantial classes.
- **Insight on inheritance quality**. The complementary perspectives on the inheritance tree provided a better understanding of inheritance relationships. We found that some superclasses were defining functionality that should be specialised by their subclasses, whereas others were defining functionality that was reused without specialisation.
- **Identification of exceptional classes**. Even if the Refactoring Browser is quite well-designed we did identify some exceptional classes that would benefit from a refactoring.
- **Overview of the methods**. We quickly identified possible singular methods and got a first view of the overall method quality. Only a few outliers possessed overly high LOC counts.
- **Internal class coupling**. Using confrontation graphs, we were able to have a fast idea of the coupling between a class and identify clusters of instance variables.

18.4 Implementation Information

CODECRAWLER is developed within the VISUALWORKS SMALLTALK environment, relying on the HotDraw framework [JOHN 92] for its visualisation. Moreover, it uses the facilities provided by the VISUALWORKS environment for the SMALLTALK code parsing, whereas for other languages like C++ and Java it relies on Sniff+ to generate code representation coded using the FAMIX Model [TICH 98](see below).

During our experiments the maximum number of entities we loaded in CODECRAWLER was 198301 (3268 classes, 35538 methods, 5420 attributes, inheritance relationships 3266, 123066 method invocations and 27743 attribute accesses). But such a limit is not linked with the approach but with the libraries used for the implementation of CODECRAWLER and the available memory.

The Underlying Data Model. CODECRAWLER is based on MOOSE a language independent representation of object-oriented source code, based on FAMIX (FAMoos Information eXchange model, see [TICH 98]) and exploits meta-modelling techniques to make the data model extensible.

A simplified view of the FAMIX data model comprises the main object-oriented concepts —namely Class, Method, Attribute and InheritanceDefinition— plus the necessary associations between them –namely Invocation and Access (see Figure 18.6).

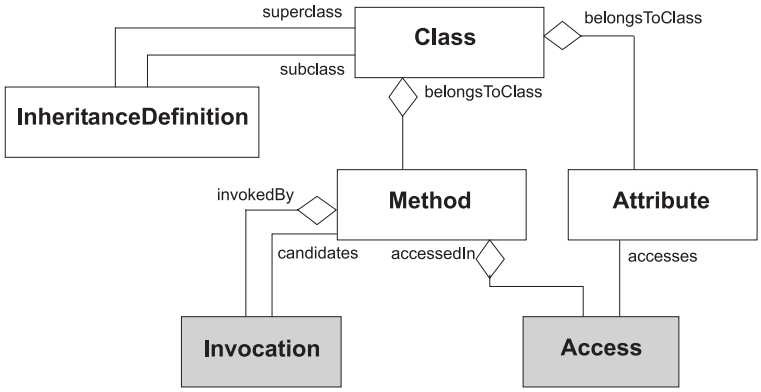


Figure 18.6: A simplified view of the FAMIX Data Model.

18.5 Contact Information

CODECRAWLER has been developed by Michele Lanza (lanza@iam.unibe.ch) during his master thesis at the University of Berne in the Software Composition Group. CODECRAWLER is based on the MOOSE language independent representation of object-oriented source code implemented in VISUALWORKS by Serge Demeyer and Stéphane Ducasse.
The authors can be contacted at {demeyer,ducasse,lanza}@iam.unibe.ch or http://www.iam.unibe.ch/scg.
CODECRAWLER is available at http:://www.iam.unibe.ch/∼scg/Archive/Software/CodeCrawler .

Chapter 19

The Nokia Environment for Re-engineering Object-Oriented Software

Author(s): Claudio Riva and Michael Przybilski

19.1 Introduction

The widespread use of object-oriented programming languages and object-oriented design paradigms is progressively increasing the amount of object-oriented software presently used in many organizations. With this a new generation of object-oriented legacy systems is emerging and technologies for dealing with their maintenance and evolution will be of vital importance in the future.

One of the main objectives of the FAMOOS project is the development of a re-engineering technology to support the evolution of object-oriented legacy systems. A basic re-engineering approach has been proposed and evaluated that consists of:

- *requirements analysis* to identify the re-engineering goals,
- *model capture* to understand its design, its architecture and its implementation,
- *problem detection* to isolate the defects,
- *problem analysis* to examine the particular problem and to select the appropriate solution,
- *reorganization* to select the optimal transformation for the system and
- *propagation* to propagate the changes in the code

To support the FAMOOS approach a re-engineering environment has been proposed [RIVA 99]. The environment consists of different tools that accomplish the different reengineering tasks. The tools assist the application of the re-engineering methodologies and automate basic and more repetitive operations. In particular, the tools are focused on the phases of *model capture*, *problem detection* and *problem analysis*.

Some of the motivations for the development of the environment were the following:

- Software organizations require their developers to evaluate the quality of their software products. The evaluation should determine if the developed software system meets the organizations quality requirements or, if these requirements are not fulfilled it should present reasons why they are not reached. Although the data regarding a software system are collected occasionally, they are seldom used for driving decisions. However, industrial experiences [ATKI 98], [AVRI 99] indicate the benefits of incorporating a quality evaluation mechanism in the standard development process and international standards [IEE 92], [ISO 98] also describe the guidelines for its application. The evaluation is valuable to the developers all along the development life cycle.
- At design time, the architects have to choose among different design alternatives and need to evaluate how a software architecture satisfies quality models [ABOW 97]. Detecting architectural shortcomings or unreliable architectures early in the life cycle is of vital importance for the organization and for the long evolution of the system. Usually these decisions are mostly based on expert judgement. Design models are the earliest product artifacts in the life cycle. As soon as design models exist, they can be evaluated by design metrics [ERNI 96] [ABRE 96] and reviewed in order to increase the software quality level [AVRI 99].
- During the implementation, developers often need to reverse engineer and reengineer parts of the developed software system. This task includes extracting information from existing code, recovering design models, understanding the dependencies between components and in general increasing the knowledge of the current implementation of the system. Also developers can benefit from evaluating the code they are writing. The evaluation should allow them to identify code that does not comply to an organization's quality levels and needs to be re-engineered. The intent is not to evaluate the programmers or their work but to provide a useful feedback to them.

The proposed environment integrates several tools using the FAMIX model [DEME 98c] as an information exchange format. The tools accomplish tasks for the analysis of source code as well as the calculation and visualization of metrics.

19.2 Description of the Environment

The front-end of the environment consists of different parsers that can be used independently. The back-end consists of a tool for metrics calculation, a UML-based visualizer and a generic graph visualizer. The integration is achieved by adopting FAMIX as a common information repository for exchanging data among the tools. This approach also allows the seamless integration of other tools that use the FAMIX model as their information source.

The parsers extract the relevant object-oriented information from the source files and store them in a FAMIX model. The information stored in the FAMIX model is sufficient for the tasks of the tools. The clients (back-end) can import the FAMIX model as a base of information for their own tasks. Figure 19.1 shows the architecture of the environment.

19.2.1 Information Exchange

The FAMIX model provides a standard mechanism for exchanging models of object-oriented systems. It is language independent, easily extensible and supports all the basic entities and relationships that are common to object-oriented languages. Entities are packages, classes, methods, attributes and other objects. Relationships are inheritance, access (a method accessing an attribute) and invocation (a method invoking a method). A FAMIX model is a representation of object-oriented source files and contains enough information for reverse engineering tasks. FAMIX models are stored using the *CASE Data Interchange Format (CDIF)* [CDI 94]. CDIF is an industrial standard for exchanging models and model instances among tools. CDIF files are stored as ASCII text.

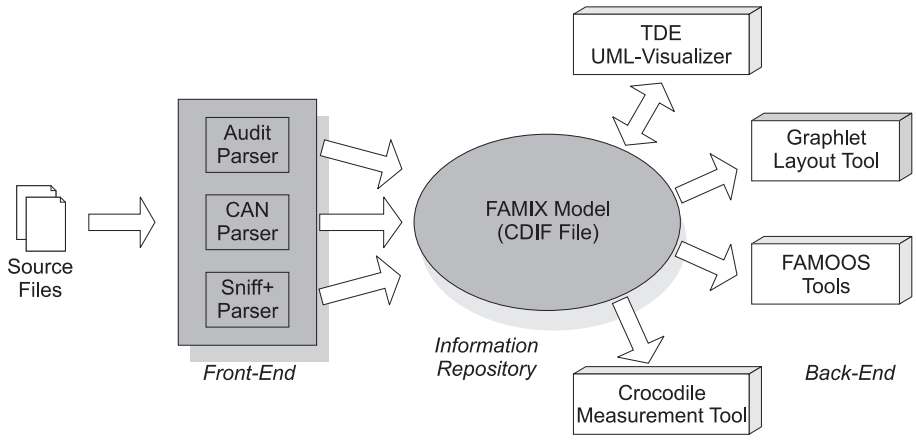


Figure 19.1: Architecture of the Environment for Re-Engineering

19.2.2 The Front-End

In order to use the environment it is necessary to gather data about the source code. This task is carried out by parsing the source files and extracting the relevant information. Although parsing in itself is well-understood, there are several problems in practice. Different dialects of the same language (C++) cause parsers that do not understand that dialect to crash or to fail. Another problem is that pre-compiler directives (like macros) or specific language constructs are usually skipped by parsers, thus causing a major loss of information. For some re-enineering tasks fuzzy parsing will extract enough information, while for others precise parsing is neccessary.

Partly because of these problems, it was decided to rely on various parsers and parsing technologies. For the proposed environment several parsers that are able to generate the CDIF files are considered.

19.2.2.1 CAN Parser

CAN [KOSK 98b] is a static C++ parser for Windows NT developed internally at the Nokia Research Center. CAN is able to parse pre-processed C++ source files according to the selected language dialect (ANSI or Microsoft-specific) and extract various object-oriented information. This information are then stored in a local relational database, whereby every input file results in a separate database.

As part of the FAMOOS project a program was implemented that is capable of exporting the information stored in the database into a CDIF file, using the FAMIX model.

The advantage of this parser is its capability to understand different C++ dialects and to use pre-processed C++ source files.

19.2.2.2 SNiFF+ Parser

SNiFF+ [TAKE 99] is produced by TakeFive Software and depicts a development environment, that is highly customizable for different programming languages like C, C++, Java or Fortran and available for different platforms (Windows, UNIX). The environment provides functionality for browsing among source files and its object-oriented entities. The source files are examined by a parser and the symbolic information is extracted and stored in a symbol table. The symbol table represents the main data repository to support the browsing capabilities. Externally the symbol table is accessible through an API.

Within the FAMOOS project, a program has been developed, capable of building a FAMIX model from the information in the SNiFF+ symbol table [TICH 99]. The program queries the data from the SNiFF+ symbol table and stores them in a FAMIX model.

The SNiFF+ environment contains several parsers for different languages, which all store their information into the SNiFF+ symbol table. This allows other object-oriented languages than C++ to be used as well. On the other hand the SNiFF+ parsers are a so-called fuzzy parser that skips pre-compiler directives, which can result in a major loss of information.

19.2.2.3 Audit Parser

Audit [Sem 99] is a tool that provides support for the static analysis of programs. Its purpose is to extract information about the programs without running them. The Audit tool contains a parser capable of extracting information from source files written in C, C++, Ada or Java. The information is stored internally in a set of tables from which it is possible to export the information into a CDIF file, using the FAMIX model.

19.2.3 The Back-End

Several CASE tools support the FAMOOS approach. Some of the tools that are capable of using FAMIX as information source and thus serving as back-end for the re-engineering environment are introduced here.

19.2.3.1 TDE: UML-Visualizer

TDE (Telecom Design Environment) is a general-purpose design tool developed at the Nokia Research Center. It mainly serves as a visual environment for the management of design information, as a graphical CASE tool that supports different design notations (like UML and Octopus) and as a collaborative workplace to share the information among a group of users. Design information is stored in a repository that can be either a shareable object-oriented database or a single user persistent object file. TDE is extensible and open in different ways: it can be interfaced with a versioning system, it can transparently access information stored in different external systems (e.g. VAX SPM, PC-LAN, World Wide Web) and an API allows a programmer to interact with its repository.

In the environment described here, TDE serves the role of an UML-based visualizer. TDE supports traditional design diagrams (like class diagrams) that conform to the UML standard. A program has been developed that enables a FAMIX model to be imported into a TDE repository. This way it is possible to visualize the class diagram of the system as represented in the FAMIX model.

The integration of TDE is useful for the architectural assessment of an object-oriented system and for reverse engineering purposes.

19.2.3.2 Graphlet: Layout Tool

Graphlet is tool capable of visualizing graphs. It is developed at the University of Passau, Germany. It allows the user to enter, edit and automatically arrange the graphs according to different layout algorithms. The graphs are saved in the GML format that is human readable and easy to generate. The tool is very useful for analyzing complex class diagrams and other graphs using different layouts.

The environment allows the user to convert a FAMIX file to the GML format in order to visualize its class diagram.

19.2.3.3 Crocodile: Measurement Tool

A software measurement tool is a necessary part to the application of a metrics approach as an integral part of the development process. The software measurement tool Crocodile is developed at the University of

Cottbus in Germany [LEWE 98b]. Crocodile enables not only calculating traditional software metrics but also supports the application of a standard software product evaluation [ISO 98].

The tool has been integrated in the Nokia environment in a cooperation between the Nokia Research Center and the University of Cottbus. Crocodile uses CDIF files that contain the FAMIX model as their information source and extracts the information necessary to calculate the software metrics from the source files. This information is stored into a local relational database that is internally used as an information repository. Crocodile allows the user to select the classes he/she is interested in and presents information about inheritance and use relationships between these classes, thus enabling the user to adapt the selection accordingly. Crocodile allows the user to select a specific quality model which is described in a configurable text file, containing metrics, how a set of metrics builds the quality model using abstractions, how these metrics are calculated and static or relative thresholds for these metrics. Crocodile computes the metrics on the selected classes according to the chosen quality model and shows the objects whose metrics do not conform to the quality model. Also graphical explanations in the quality tree and textual explanations about such outliers is given.

As part of the integration into the re-engineering environment, Crocodile was modified to be able to visualize its result within TDE. In particular the selected classes and the calculated measures are exported to a HTML file. In TDE's class diagram, the classes that violate the metrics are marked with red and a link to the HTML file is attached to them (see Figure 19.2).

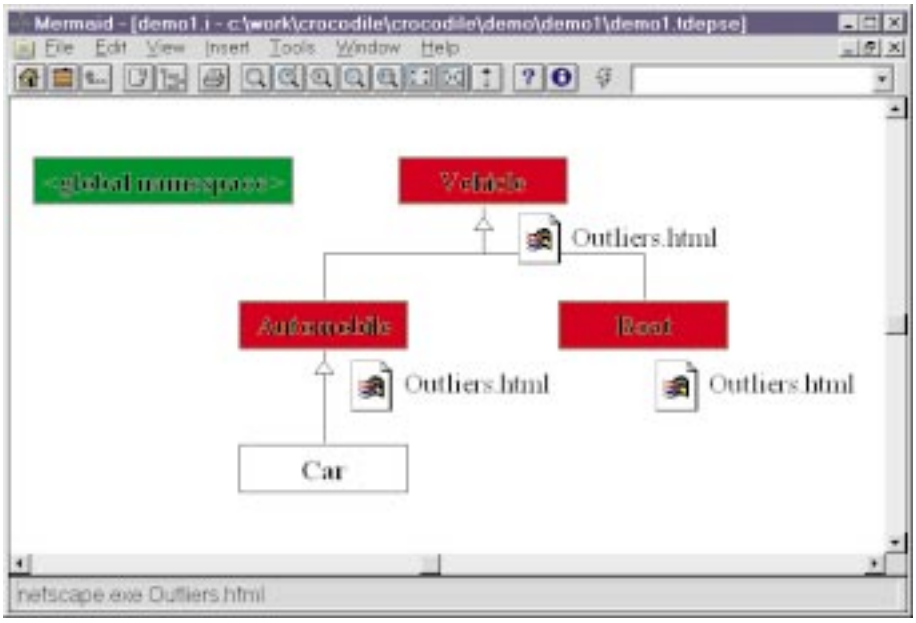


Figure 19.2: A Class Diagram with Outliers in TDE

This way it is possible to browse a class diagram and recognize outliers instantly. Then it is possible to use the attached link to observe the calculated metric values in a generic HTML browser.

19.2.3.4 FAMOOS Tools

Several tools with re-engineering capabilities have been developed during FAMOOS, several of which are able to import or generate FAMIX files. They provide several reverse-engineering capabilities such as automatic problem detection, software measurement or reorganization capabilities.

19.3 Scenarios of Use

The presented re-engineering environment can be used during most parts of the reverse-engineering process. Some of the scenarios in which the environment could be used are presented here.

19.3.1 Analyzing Legacy Code

The coexistence of legacy code within new systems raises issues that are well known by the research community [ARNO 92]. Often, such coexistence requires programmers to reverse engineer old or badly documented parts of the system. The environment automates the task of extracting object-oriented entities from the source code and presenting them in UML class diagrams. Such diagrams are useful to keep the design documentation up to date or to re-document the software system. Metrics also help the developer in localizing the parts of the system that need to be re-engineered.

19.3.2 Ensuring Quality Requirements

A software developer can use the environment to ensure his program meets the quality requirements of the customer. Detecting the outliers with the quality model and visualizing them within a class diagram depict an attractive way to examine metrics values. Also analysts can benefit from this feature, since they can examine how outliers are distributed within the system architecture. The environment enables programmers to monitor the changes in the quality of their code as the development proceeds.

19.3.3 Conventional Reverse Engineering using Multiple Front-Ends

The environment supports different parsers that adopt different parsing technologies (fuzzy parser, exact parser) and support several languages (C++, Java, Ada, etc.). One advantage of the environment is that it provides the user the extensibility and flexibility to choose and integrate other parsers to extract the data from the source files.

19.4 Evaluation of the Environment

The environment was evaluated on a real case study [CZAR 99]. In this section some examples of its usage are presented, focussing on the phases of model capture and problem detection of the FAMOOS approach. The evaluation followed four basic steps:

1. Building the FAMIX model by extracting the information from the source files.
2. Calculating several software metrics.
3. Examining the metrics to detect their outliers.
4. Analyzing the outliers in the class diagram.

A large software system developed in C++ was used as the case study of this experiment. It consists of about 1 MLOC (including comments) and contains about 2000 classes. SNiFF+ was used to parse the case study and to generate the FAMIX model. As samples two software metrics were calculated: the Number of Children (NOC) and the Number of Public Methods (NOPM). Only one outlier for each metric was selected for analyzation in a class diagram. It must be pointed out that the detection of metric outliers does not lead to the identification of problems but it just provides hints where to start the examination. The purpose of this short demonstration is to show how the tools support the FAMOOS approach.

19.4.1 Number of Children

This scenario considers the analysis of the amount of children of each class. Figure 19.3 shows the cumulative distribution of the metric.

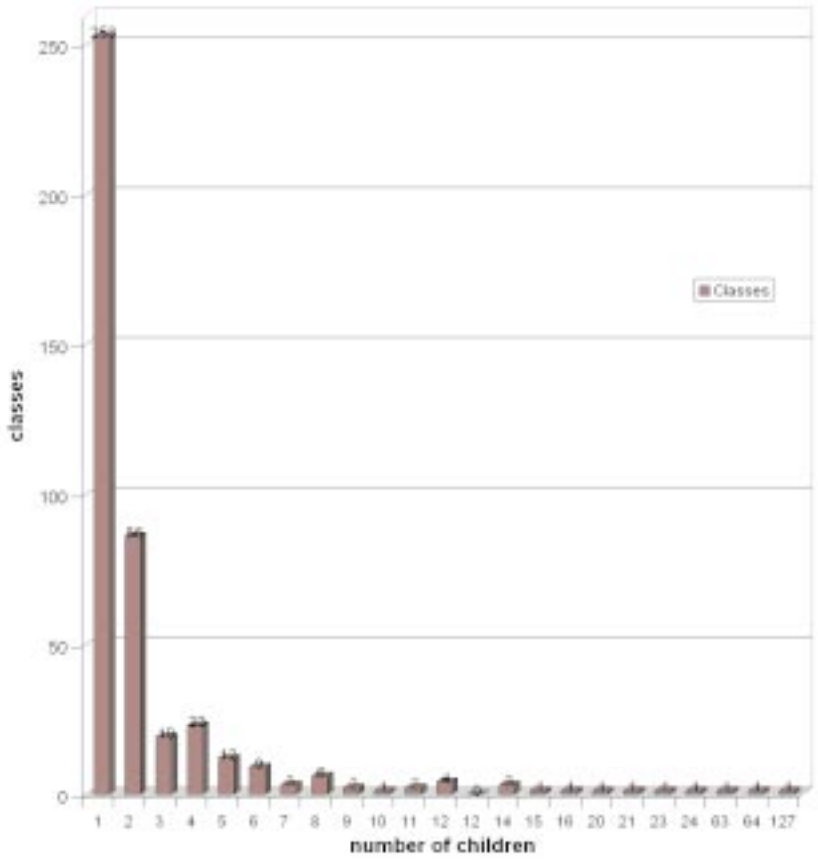


Figure 19.3: Number of Children

It shows 1877 classes without any child. The outliers of this metric are located on the right of the graph. In particular the attention was focused on the class with 127 children. Graphlet was used to layout the class diagram of the class and its descendents. It is shown in Figure 19.4.

The central class depicts the root of the tree. The first ring of classes depicts the 127 children of the root class. The architecture seems to be repetitive and additional levels of inheritance could simplify its design.

19.4.2 Number of Public Methods

The scenario considers the analysis of the number of public methods for each class of the system. The cumulative distribution is represented in Figure 19.5.

There are 651 classes without any methods. These depict mainly *struct* constructs or simple data structures. Most of the classes have less than 20 methods, which is a good design choice since it keeps the complexity of the classes low. There are also several classes with a high amount of methods. Their class diagram was analyzed and can be seen in Figure 19.6.

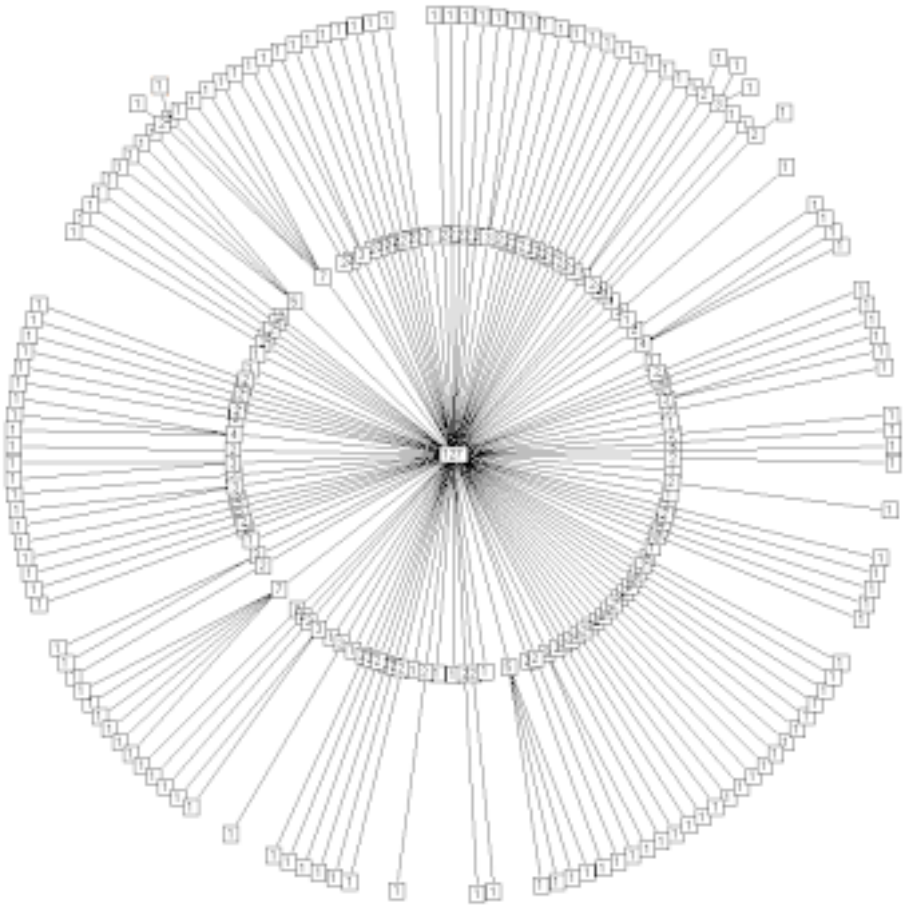


Figure 19.4: Class Diagram with many Children

At the top of the diagram there are classes with many public methods and which use multiple inheritance. The descendents inherit the high complexity of the parents, thus becoming more difficult to understand. Although the design of the diagram is simple, the complexity behind its classes is very high. The reasons for this design have to be searched in the tasks of the classes.

19.5 Conclusions and Future Work

In this document an environment for re-engineering object-oriented software was presented. The environment consists of tools for parsing, measuring and visualizing object-oriented software. The mechanism for exchanging information is based on the FAMIX model that has been developed within the FAMOOS project.

Future work will concentrate on extending the environment. In particular work will focus on the following topics:

- customization and extention the default quality model according to the organization's quality parameters.
- improving and developing other means for presenting the metrics within a class diagrams or with other graphical techniques.
- improving the graphical layout of the diagrams visualized in TDE.

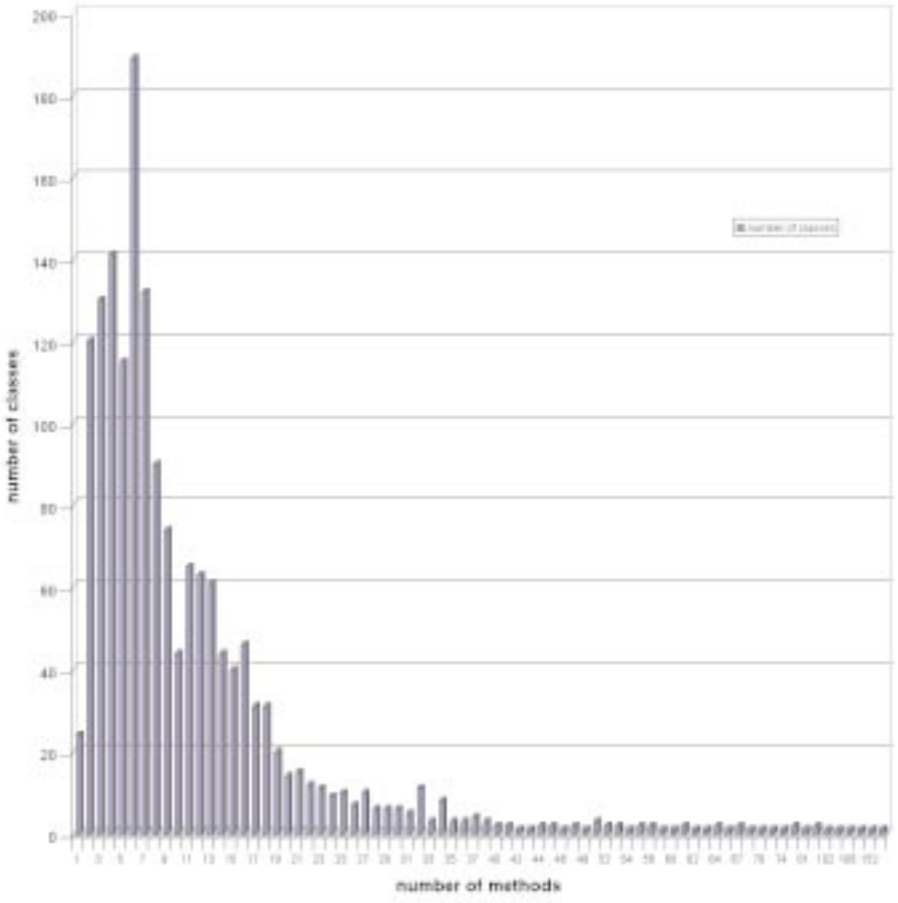


Figure 19.5: Number of Methods

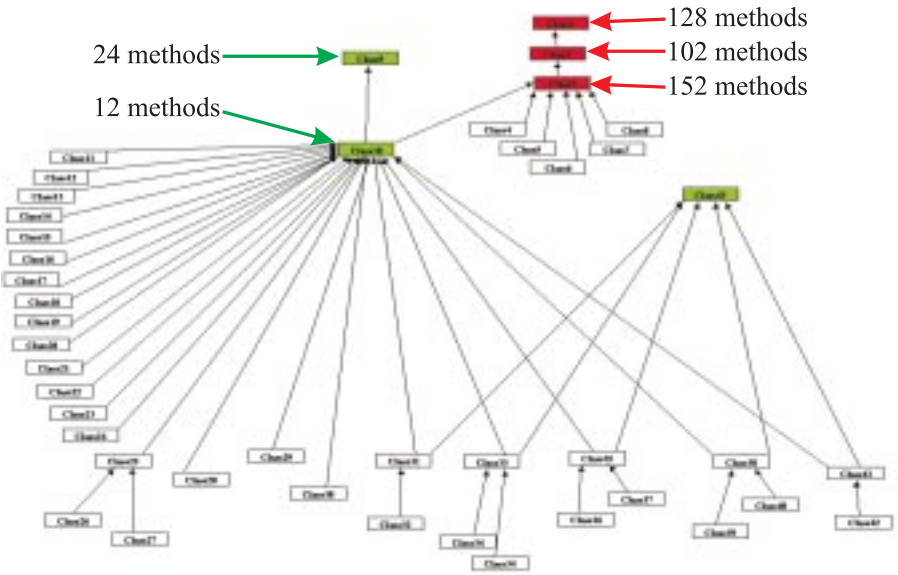


Figure 19.6: Class Diagram of Classes with many Methods

Part V

Background

Chapter 20

Metrics

Author: Radu Marinescu

20.1 Introduction

Software engineering involves the study of the means of producing high quality software products with predictable costs and schedules. One of the major goals in software engineering is to control the software development process, thereby controlling costs and schedules, as well as the quality of the software products. As a direct result of software engineering research, software metrics have been brought to attention of many software engineers and researches. As De Marco points out, "you cannot control what you cannot measure" [DEMA 82]. Software metrics measure certain aspects of software and can be generally divided into three categories [DUMK 97]:

- Process measurement for understanding, evaluation and improvement of the development method
- product measurement for the quantification of the product (quality) characteristics and the validation of these measures
- resource measurement for the estimation of needed resources in terms of human and hardware resources

This work focuses on software products metrics.

The software metrics for the procedural paradigm have concentrated on measures of complexity and have used different aspects of the source code of the software been analysed. Some count certain lexical tokens [HALS 77], [BAIL 88]or are based on its control graph [MCCA 76] Another set of metrics measures the inter-connection among statements or functions [HENR 81], [MCCL 78], [WOOD 80], [ADAM 90], [ROBI 89].

Since the OO paradigm exhibits different characteristics from the procedural paradigm, software metrics need to take them in account. The OO software metrics need to consider the basic concepts of the OO model as: object, class, attributes, inheritance, method and message passing.

So far, few object oriented metrics have been proposed, and few studies to validate them were made, further work in this direction needed to be done.

20.1.1 Outline

This chapter is based on [MARI 97] and organised as follows: Section 20.2 provides an overview of the most important theoretical foundations in software measurement, with a special focus on measurement in the object oriented approach. Some fundamental theoretic notions from the measurement theory are briefly discussed. The first section of this section also includes a summary of the representational theory of measurement. The middle section of this section offers a ontological basis for the key concepts of the object oriented approach that the most of the metric definitions deal with. We conclude this section by discussing the possibilities of establishing a set of criteria for the analytical evaluation of object oriented metric definitions.

In section 20.3 we present the important object oriented metrics found in the literature research. The metrics are synthesised in four sections, corresponding to the main aspect measured by each of them. Firstly we will refer to metrics that are mainly determining the level of class complexity; than we will present a couple of metrics that measure the quality of the class hierarchy layout. Next a set of metrics referring to coupling will be discussed; and finally some metrics that refer to the aspect of cohesion will be presented. After discussing the theoretical aspects about these metrics, we conclude the section looking at some experimental results obtained by other researchers in using some of the metrics discussed here.

Section 20.4 presents the set of metrics that was selected for the experimental study, explaining the criteria that determined their selection. Additionally, we present some experiences we made while applying these metrics to casestudies. We investigate the efficiency of using these metrics from the perspective of a redesign process, on a number of three object oriented systems. For each metric we present the obtained results, and than we give an interpretation of these results and try to make an evaluation of the practical utility of that metric.

Based on the experiences we made by using metrics in reengineering projects, we defined some new metrics to measure the amount of reuse in class hierarchies that comes from inheritance. These metrics are presented in section 20.5.

20.2 Foundations of Software Measurement

This section presents some of the most important elements of the software measurement theory. This section consists mainly of two parts. In the first part some general fundaments from the theory of measurement are briefly introduced, including a description of the Representational Theory of measurement. The second part of this section focuses on the matter of the theoretical assessment and validation of metrics, explaining the need for such a validation and introducing the main contributions towards the definition of a rigourous framework for software measurement.

20.2.1 Fundaments of the Measurement Theory

Software measurement, like measurement in any other discipline must adhere to the science of measurement if it is to gain a widespread acceptance an validity. Measurement theory is used to highlight both weaknesses and strengths of the software metrics work. The theory of measurement helps us to define and validate measures of specific complexity attributes. In this section we will describe some general fundaments from the theory of measurement. In the first part of this section, we will give some definitions and two possible classifications of the measurement process. The other part of this section describes the representational theory of measurement. This section is mainly based on the considerations of Fenton [FENT 94]. Most of the details concerning scale types rely on the approach of Zuse [ZUSE 95].

20.2.1.1 Definition and Classification of Measurement

Definition of Measurement. We define *Measurement* as the process by which the numbers or symbols are assigned to attributes of entities in the real world in such way as to describe them according to clearly defined rules.

This definition of measurement requires some explanations and some other definitions. The concepts used in this definition, like: entity, attribute, rules will be next defined.

Definition 1 *We define an* entity *as the subject of the measurement process. An entity might be an object, or a software specification or a phase of a project. An* attribute *is a feature or property of the entity. For example, an attribute of a software specification is its length, and an attribute of a project phase may be its duration. In most situations an attribute may have different intuitive meanings to different people. Thus we have to define a* model *as the expression of a viewpoint concerning the entity being measured.*

Once a model has been chosen it becomes also possible to determine relations between the attributes that describe the entity being measured. The need of defining good models is particularly relevant in software engineering measurement. For example, even a simple measure like the length of a program requires a well defined model of programs which enables us to identify unique lines unambiguously. Similarly, to measure the effort spent on a phase of a project needs a clear "model" of the phase which at least makes clear when the process begins and ends.

Classification of Measurement Activities. We will classify the measurement activities using two different criteria. The first classification uses as criteria the way the measurement is realised. Corresponding to this type of classification there are two type of measurements:

- **Direct Measurement** of an attribute is a measurement that does not depend on the measurement of any other attribute. In most of the cases the direct measurement is more simple.
- **Indirect Measurement** of an attribute is the measurement that involves the measurement of one or more other attributes. The indirect measurement is normally more sophisticated.

The second classification analyses the problem from another viewpoint. The criteria used for this second classification is the use of measurement. Corresponding to this criteria there also two broad uses of measurement:

- **Assessment Measurement** of an attribute is a measurement that determine the current measure of an attribute, that is the value of the attribute at the present moment of time.
- **Predictive Measurement** of an attribute is the measurement that depends on a mathematical model relating A to some existing measures of $A_1, \ldots, A_n$. It gives a prediction of the future measure of A based on the present measures of $A_1, \ldots, A_n$.

20.2.1.2 The Representational Theory of Measurement

Although there is no universally agreed theory of measurement, most approaches are devoted to resolving the following three main issues:

- Which types of attributes can be measured? (representation problem)
- How to define measurement scales? (scales and scale types)
- What kind of statements about measurements are meaningful? (meaningfulness)

In this section we will present a brief overview of the representational theory of measurement, pointing to the manner in which this theory resolves the three main issues mentioned before.

Empirical Relation System. Direct measurements of a particular attribute possessed by a set of entities must be preceded by intuitive understanding of that attribute. This intuitive understanding leads to the identification of the empirical relations between the entities.

Definition 2 *The set of entities C, together with the set of empirical relations R, is called an empirical relation system (C, R) for the attribute in question.*

Example: The attribute of "height of people" gives rise to empirical relations like "is tall", "taller than" and "much taller than".

Representation Condition. To measure the attribute that is characterised by the empirical relation system (C, R) requires a mapping M into a numerical relation system (N, P). Specifically, M maps entities in C to numbers (or symbols) in N, and empirical relations are mapped to numerical relations in P, in such way that all the empirical relations are preserved. This is the so-called *representation condition*, and the mapping M is called *representation*. Formally we may express the representation M as follows:

$$M : (C, R) \rightarrow (N, P)$$

Definition 3 *The representation condition asserts that the correspondence between empirical an numerical relations are is two way. Suppose for example the binary relation is mapped by M to the numerical relation $<$. Then formally, we have the following expression of the representation condition:*

$$\forall x, y \in C \wedge' \prec' \in R, \exists' <' \in P : x \prec y \Leftrightarrow M(x) < M(y)$$

Example: We may extend the example from the previous section. Suppose C is the set of all people and R contains the relation "taller than". A measure M of height would map C into the set of real numbers $\mathcal{R}$ and 'taller than' to the relation '$>$'. The representation condition asserts that person A is taller than person B if and only if $M(A) > M(B)$.

Scale. Representation Problem. Based on the aspects defined and discussed previously in this section we can give now the definition of the scale:

Definition 4 *Being given an empirical relation system $E = (C, R)$, a numerical(formal) relation system $F = (N, P)$ and a measure $M : E \rightarrow F$, the triple (E, F, M) is called a scale.*

Having defined all these elements, we can look back at the issues that should be solved by a theory of measurement, and we can already answer the first question: which types of attributes can be measured? This question can be expressed formally as follows: being given an empirical relation system E and a formal relation system F, the question that is raised is if there exists for this two relation systems a representation so that (E, F, M) is a scale. This problem is called the representation problem. If such a measure exists the attribute described by the empirical relation system E is measurable. In other words, an attribute of a set of entities, is measurable if and only if it is possible to find a representation that satisfies the representational condition.

Admissible Transformation. If the representation problem is solved, we can examine the next issue. Supposing that for an empirical relation system E and a formal relation system F, we have found a measure, we can ask how uniquely this measure is. Suppose that an attribute of some set of entities has been characterised by an empirical relation system (C, R). There may in general be many ways of assigning numbers which satisfy the representation condition. For example, if person A is taller than person B, then $M(A) > M(B)$ irrespective of whether the measure M is in inches, feet, centimeters, etc. Thus, there are many different measurement representations for the normal empirical relation system for the attribute of

height of people. However, any two representations M and M' are related in a very specific way: there is always some constant $c > 0$ such that: $M = cM'$. This problem is called the uniqueness problem. This uniqueness problem leads us to the definition of scale types. In order to give the definition of a scale type, we have to define what is an admissible transformation.

Definition 5 *The transformation from one valid representation into another is called an admissible transformation. Formally, this definition can be expressed as follows: let (E, F, M) be a scale. A representation M' is an admissible transformation, if (E, F, M') is also a scale.*

Scale Types Hierarchy. There are different scale types and they can be ordered in a hierarchy that reflects the richness of knowledge concerning the empirical relation system. For example, where every admissible transformation is a scalar multiplication the scale type is called ratio. The ratio scale is a sophisticated scale of measurement which reflects a very rich empirical relation system. An attribute is never of ratio type a priori.

In table 20.1 the hierarchy of the best known scale types is presented, in the increasing order of sophistication. We normally start with a crude understanding of an attribute and a means of measuring it. Accu-

Name of the scale type	**Type of admissible transformation** (f)
Nominal Scale	any one-to-one $f(x)$
Ordinal Scale	$f(x)$ - strictly increasing function
Interval Scale	$f(x) = ax + b, a > 0$
Ratio Scale	$f(x) = ax, a > 0$
Absolute Scale	$f(x) = x$

Table 20.1: Hierarchy of scale types. The lowest scale is the nominal and the highest one is the absolute scale.

mulating data and analysing the results leads to the clarification and revaluation of the attribute. This in turn leads to refined and new empirical relations and improvements in the accuracy of the measurement; specifically this is an improved scale.

For many software attributes we are still at the stage of having very crude empirical relation systems. In the case of an attribute like "criticality of software failures" an empirical relation system would at best only identify different classes of failures and a binary relation "is more critical than". In this case any tow representations are related by a monotonically increasing transformation. With this class of admissible transformations, we have an ordinal scale type.

Meaningfulness. If we know the scale type, this enables us to determine rigorously what kind of statements about measurements are meaningful, and which are not.

Definition 6 *A statement involving measurement is meaningful if its truth or falsity remains unchanged under any admissible transformation of the measures involved.*

Meaningfulness guarantees that the truth values of statements with measurement values is invariant to scale transformations.

Example: It is meaningful to say that "A is twice as tall as B"; this statement is true/false if we express the heights in meters and it will still remain true/false if we express the height in inches, or in any other measurement unit. On the other hand the statement "Failure x is twice as critical as failure y" is not meaningful if we only have an ordinal scale empirical relation system for failure criticality. This is because a valid

ordinal scale measure M could define $M(x) = 6$, $M(y) = 3$, while another valid ordinal scale measure M' could define $M'(x) = 10$ and $M'(y) = 9$. In this case the statement is true under M but is false under M'.

The notion of meaningfulness also enables us to determine what kind of operations we can perform on different measures. For example, it is meaningful to use means for computing the average of a set of data measured on a ratio scale, but not on a ordinal scale. Medians are meaningful for an ordinal scale but not for a nominal one.

20.2.2 Theoretical Validation of Software Measurements

Software measurement must adhere to the science of measurement if it is to gain a widespread acceptance. Consequently, it is essential that the measures we define and use are valid. In other words, the measures we use must accurately reflect the attributes they are supposed to quantify.

20.2.2.1 The Necessity

There are a number of serious reasons that motivate a rigourous theoretical validation:

1. Many times we find in the literature multiple metrics measuring the same attribute in different manners and offering contradictory results. This is mainly because the measures do not properly capture the attribute to be measured [FENT 97] [FENT 94].
2. Often measures are not accepted in the industry because the concepts they work with are not precisely defined. [BRIA 96a]
3. A common practice is that of "validating" a new measure by empirically finding a correlation with a commonly accepted measure. This is unacceptable as long as no more solid basis for this correlation can be found. [HEND 96]
4. Empirically defined measures are "highly parochial, highly limited and highly unscientific" [HEND 96]. Thus, their use is restricted to the strict context in which they are defined and their proper use in other contexts is extremely reduced.
5. A number of *dimensional inconsistencies* often appear in metrics definitions [HEND 96]. The most common inconsistencies are: unmatched dimensions on the right-hand and the left-hand side of an equation; addition of metrics defined on different scales; use of meaningless operations (a kind of adding oranges and apples). This implies the need of a *dimensional analysis*. [KITC 95].

20.2.2.2 Weyuker's Properties for Complexity Measures

In a notable paper [WEYU 88] Weyuker proposed a set of nine axioms against which software complexity metrics could be evaluated formally. It identifies a set of desirable properties a measure should posses. Although the paper is important, it was strongly criticised, most notable by Zuse [ZUSE 90] and Fenton [FENT 97] [FENT 94].

The Properties. Weyuker's first four properties address how sensitive and discriminative the measure is. The *first property* ensures that any measure that rates all programs to be of the same complexity is eliminated from consideration. The *second property* extends the concept of the first one ensuring that there is sufficient resolution in the measurement scale to be useful (e.g. a measure with only two values is unlikely to have a large applicability). The *third property* counterbalances the second one by requiring that the metric should not be so fine that any specific value of the metric is only achieved by a single

program. The *fourth property* suggests that it is possible for the same functionality to be encapsulated in two different programs with different complexities. The *fifth property* requires that the value of a measure for a component of a program is no more than the value produced for the whole program. The *sixth property* implies that by concatenating a program R with each of two equally complex programs, P and Q, it is not necessarily the case that the resulting complexity after merging would be identical. The *seventh property* requires for a measure to be sensitive to statement order within a program. The *eighth property* states that renaming of variables should not affect the value of a measure. Finally, the *ninth property* requires that the sum of the complexities of the components of a program could be less than the complexity of the program when considered as a whole.

Criticisms. Fenton [FENT 97] notes that the underlying problem with Weyuker's nine axioms is that "complexity" is ambiguously defined and it can be interpreted to mean size, comprehensibility, ease of implementation etc. Fenton's main concern lies with the fifth and sixth properties, because the fifth property is appropriate for complexity measures related to size but not those relating to comprehension, while the sixth property is appropriate for comprehensibility measures, but is not appropriate for size measures.

Zuse [ZUSE 90] notes that these properties are not in accord with measurement theory since they require a ratio scale in parts and reject the ratio scale in other places. He affirms that the ninth property is meaningful on a ratio scale but not on an interval scale. On the other hand, the sixth and seventh property deny the ratio scale.

Kitchenham et al. [KITC 95] go a step further than Zuse and reject Weyker's fifth, sixth and ninth property because they imply a scale type at all. They also suggest that the seventh property is inappropriate because it contradicts standard measurement practice, and discard the eight property as unnecessary if accepting the Representation Condition. The rest of the properties are assimilated by Kitchenham et. al. and included in their framework [KITC 95].

20.2.2.3 The Kitchenham-Pfleeger-Fenton Framework

The framework proposed by the three authors [KITC 95] is based on identifying the elements of measurement and their properties (structure model), identifying how these elements are defined when constructing a measure (definition models), and defining appropriate theoretical and empirical methods of validating the properties and definition models. This framework was already applied in [HARR 98] for the validation of the MOOD metrics suite [BRIT 95].

Structure Model. The structure model describes the objects involved in measurement and their relationships. The objects identified in this model are: entities (objects from the real world), attributes (properties of an entity), units (mappings of attributes to the formal world), scale types, values (a specific measurement unit applied to a particular entity and attribute) and measurement instruments (instruments used to obtain the values). The relationships among these objects (for the case of a direct measurement) are depicted in Figure 20.1 .

Definition Models. The framework of Kitchenham et al. presents models for defining units, instruments, attribute relationships, measurement protocols and entity populations. The authors identify four types of units definitions: by reference to a standard (example), by reference to a wider theory, by means of conversions from other units or by using a model involving several attributes. Concerning instruments, two types of instrumentation models are identified: a direct, representational model and an indirect, theory-based one. Attribute relationship models are used when considering an attribute derived from a functional relationship involving one or more other attributes. In conformity to this framework there are two types of attribute models: definition and predictive models. Measurement protocols are concerned mainly with *how* an attribute can be measured consistently on a specific entity. The requirements for any measurement protocol is to be unambiguous, self-consistent and to prevent invalid measurement procedures (e.g. double counting).

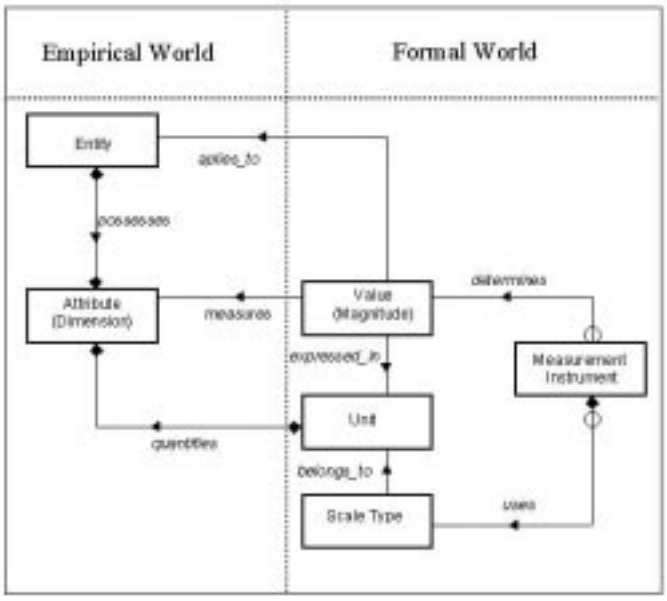


Figure 20.1: Relationship among objects involved in a direct measurement

Properties of Measures. Kitchenham et al. identify a number of theoretical criteria that need to be satisfied by any valid measure:

1. For an attribute to be measurable, it must allow different entities to be distinguished from one another.
2. A valid measure must obey the Representation Condition
3. Each unit of an attribute contributing to a valid measure is equivalent.
4. Different entities can have the same attribute value (within the limits of measurement error).

In addition to the four properties stated before, indirect measures must exhibit a number of supplementary properties. Thus, an indirect measure must:

- be based on a model concerning the relationship among attributes;
- be based on a dimensional consistent model;
- exhibit no unexpected discontinuities;
- use units and scale types correctly

20.2.2.4 Property-Based Software Measurement

As we mentioned in the beginning of this section, many times measures defined to capture the same attribute offer contradictory results, mainly because measures do not properly capture the attribute to be measured. Thus, wide accepted sets of properties for different internal attributes like coupling, cohesion or complexity are strongly needed. In order to achieve this goal concepts must be clearly separated and their similarities and difference studied. The paper of Briand et al. [BRIA 96a] aims to define concepts through different sets of unambiguous and intuitive properties that characterise following internal attributes: size, length, complexity, coupling and cohesion. Below we present the properties defined for each of these attributes.

Size. According to this framework size cannot be negative (Size 1), and is expected to be null when a system does not contain any elements (Size 2). When modules do not have shared elements the size is expected to be additive (Size 3). Two corollaries derive from these properties: the size of a system is not greater than the sum of sizes of any pair of its modules (Corollary 1); the size of a system built by merging modules cannot be greater than the sum of the sizes of the modules (Corollary 2).

Length. Length is defined as the maximum distance between two points (elements). Length is non-negative (Length 1) and equal to 0 where there are no elements (Length 2). If a new relationships is introduced between two elements belonging to the same connected component of a system graph, the length of the new system cannot increase (Length 3). If a new relationship is introduced between two elements not belonging to the same connected component of the system graph, the length of the new system cannot decrease (Length 4). Length is not additive for disjoint modules; it is the maximum length among the modules (Length 5).

Complexity. This concept is defined as a property of a system that depends on the relationships between the elements. Complexity is expected to be non-negative (Complexity 1) and to be null (Complexity 2) when there are no relationships between the elements of the system. Complexity should not be sensitive to the convention chosen to represent the relationships between the elements (Complexity 3). Complexity of a system should be at least as much as the sum of the complexities of any collection of its modules (Complexity 4). A consequence of the above property is that the complexity of a system should not decrease when the set of relationships increases (Corollary 1). The complexity of disjoint modules is additive (Complexity 5).

Cohesion. Cohesion is defined as the tightness with which related program features are grouped together in systems or modules. Cohesion is expected to be non-negative and *normalised* (Cohesion 1) so that the measure is independent of the size of the module. If there are no internal relationships in a module, cohesion should be null (Cohesion 2). Additional internal relationships in modules cannot decrease the cohesion of the module (Cohesion 3). When two or more non-related modules are merged, cohesion cannot increase (Cohesion 4).

Coupling. The concept of coupling refers to modular systems and it captures the amount of amount of relationship between the elements belonging to different modules of the system. Coupling is also expected to be non-negative (Coupling 1) and null when there are no relationships among the modules (Coupling 2). Coupling does not decrease when additional relationships are created across modules (Coupling 3). Merging modules can only decrease coupling (Coupling 4). Finally, coupling is additive for disjoint modules (Coupling 5). The authors also discuss some of the best-known measures in the context of the proposed set of properties verifying if the metrics are indeed measuring the attributes they claim to measure.

20.3 A Survey of Object Oriented Metrics

This section offers an overview of the most important object oriented metrics defined in the software metrics literature. The metrics presented in this section are classified in four different sections: complexity metrics, metrics measuring the class-hierarchy layout, metrics that measure the coupling between classes, and finally a set of metrics that measure the cohesion of a class. For each analysed metric the definition of the metric is provided together with its main characteristics. For the most of the metrics the critics found in the researched literature are also discussed. A final section points to some of the practical experiments that were made using some of the metrics presented in this section.

20.3.1 Class Complexity Metrics

By complexity metrics we consider those metrics that may give us indications about the level of complexity for a given class. Analysing the several viewpoints suggested for each metric, it becomes clear that we generally expect from these metrics to be predictors of the maintenance effort for a class. They may also be an indicator of the central classes in a project. We may also say that the place of the complexity metrics in the object oriented design (OOD) process may be the phase of identifying the semantics of classes, and some of the metrics may be used also in the class identification phase.

20.3.1.1 Weighted Method Count – WMC

Definition 7 *Consider a class $C1$, with methods M_1, M_2, ... , M_n that are defined in the class. Let c_1, ... , c_n be the complexity of the methods of that class. Then:*

$$WMC = \sum_{i=1}^{n} C_i$$

Concerning the way this metric defined, there are two things that should be observed. The first observation that should be made is that the notion of complexity is deliberately not defined more specifically in order to raise the generality degree of the WMC metric. The way complexity is specifically defined for an implementation of this metric is a decision that can taken in different ways. One possible direction is the use of some traditional static complexity metrics, like the McCabe's Cyclomatic Complexity Metric. This was suggested by Li and Henry in [LI 93].

The second observation is that if we consider each metric having a unitary complexity, then WMC = n, the number of methods in that class. That means that if we consider all the methods of the class having equal complexity, this metric indicates the number of methods of the class. From another perspective this metric can be seen as a generalisation of the Number of Methods metric.

Viewpoints. Chidamber and Kemerer give also a set of three viewpoints that should be used in interpreting the results of this metric:

1. The number of methods and the complexity of methods involved is a predictor of how much time and effort is required to develop and maintain the class.
2. The larger the number of methods in a class the greater the potential impact on the children, since the children will inherit all the methods defined in the class.
3. Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

Critics. The WMC metric is widely accepted in the literature. Yet it has been indirectly criticised by Churcher and Shepperd in [CHUR 95]; The critic was referring to the computation of the number of methods in a class. The number of methods is required directly for the computation of WMC, as we already pointed above. The main thesis of Churcher and Shepperd is that the computation of the number of methods in a class is open to a variety of interpretations. For example they suggest that overloaded constructors, that share of course the same name identifier should not be counted distinctly. Another suggestion the two authors make is that the inherited methods should also be counted. In fact there are many different possibilities of counting methods:

- when developing metrics concerned with services offered by an object, it is likely that only distinct method names will be considered

- service-oriented metrics may omit methods that implement operators, since they are seldom invoked explicitly
- metrics aimed at predicting maintenance effort are more likely to include all the constructors separately by counting signatures
- to understand what a class does, the most important source of information is its own operations. Therefore, restrict counting to the current class, ignoring inherited members is another possibility, sincethe class increment is the best measure of its functionality
- the approach of counting including all inherited methods emphasises the importance of state space, rather than the class increment, in understanding a class. But if the inherited methods are included, then features of derived class can be obscured
- allowing different strategies for each level of inheritance further increases the number of potential counting strategies

Chidamber and Kemerer replied to these critics in [CHID 94] by referencing the principle of counting the methods in a class: "the methods that required additional design effort and that are defined in the class should be counted, and those that do not should not."

However we think, that there is no "correct" way to count methods; rather, the precise form appropriate to particular circumstances must be specified, mainly where the quantities measured are then used to derive metrics indirectly depending on the number of methods.

20.3.1.2 Response For a Class – RFC

Definition 8 *The Response Set for a Class (RS) is a set of methods that can be potentially executed in response to a message received by an object of that class. Mathematically it can be defined using elements of set theory, as:*

$$RS = \{M\} \cup_i \{R_i\}$$

where $\{R_i\}$ *is the set of methods called by method* i *and* $\{M\}$ *is the set of all methods in the class.*

The Response for a Class (RFC) is the cardinality of the response set for that class. Mathematically the RFC metric, can be expressed as:

$$RFC = \mid RS \mid$$

The response for a class (RFC) metric measures two different aspects: On the one hand it measures the attributes of the objects in the class; on the other hand it expresses the potential communication between the class that is measured and the other classes, as it includes method called from the outside the class.

Viewpoints. Chidamber and Kemerer suggests also a set of three viewpoints that should guide the interpreting and use of the results of this metric:

1. If a large number of methods can be invoked in response to a message the testing and debugging of the class becomes more complicated since it requires a greater level of understanding required on the part of the tester
2. The large number of methods that can be invoked from a class, the greater the complexity of that class.
3. A worst case value for possible responses will assist in appropriate allocation of testing times.

We have to observe that the three viewpoints are not independent, but logically derived. The first and the third viewpoints are conceptually derived form the second viewpoint. In the same time the first and third viewpoint are very similar. This affirmation is based on the fact that high complexity logically implies complicated - and therefore time-expensive - testing and debugging phases.

Critics. Referring to the RFC metric, Hitz and Montazeri in [HITZ 96b] give the following alternative definition of the metric: "RFC is the union of the protocol a class offers to its clients and the protocols it requests from other classes." Their critic towards this metric refers to the fact that by measuring the total communication potential, this measure is not only a measure of the complexity of the class but it is obviously related to coupling and by this it is not independent of the CBO metric, also defined by Chidamber and Kemerer[1]

20.3.1.3 SIZE1 and SIZE2

This two size metrics were defined by Li and Henry in [LI 93]. Size has been used as a software measure for a long time. The Lines of Code (LOC) metric is used to measure a procedure or a function and the accumulated LOC of all procedures and functions for measuring a program.

Definition 9 *The SIZE1 metric is in fact the traditional LOC metric which is calculated by counting the number of semicolons in a class. Thus the LOC metric is referred by Li and Henry as SIZE1:*

$$SIZE1 = \text{number of semicolons in a class}$$

The second size metric defined in the mentioned paper is the number of properties, including the attributes and methods defined in a class. This metric is referred as SIZE2 and is calculated as follows:

$$SIZE2 = \text{number of attributes} + \text{number of local methods}$$

Critics. The critic of SIZE1 is that it is a metric defined for the traditional approach and, as the author himself underlined the size factor in an object oriented program has not been well established. Another critic, on the use of size metrics in the object orient approach comes from Stiglic, Hericko and Rozman. According to them: "size metrics such LOC and Number of Classes are not very useful and obviously have no relation with quality attributes." [STIG 95]. Also, these metrics were not discussed or analysed in other papers and even the authors have concluded that the two metrics alone do not give useful information in predicting maintenance efforts.

20.3.2 Measuring the Class Hierarchy Layout

This category of metrics tries to give indications on the quality of the class hierarchy layout of a project. The quality of the class hierarchy plays a major role in the design of an object oriented system. It is clear that this kind of metrics could be very important indicators in the early phases of the design, i.e. in the classes identification phase, where the possible classes of the project are being identified.

20.3.2.1 Depth of Inheritance Tree – DIT

Definition 10 *Depth of Inheritance Tree (DIT) represents the length of the tree from that class to the root class of the inheritance tree. In cases involving multiple inheritance, the DIT value will be maximum length from that node class to the root class of the tree.*

An observation should be made, concerning this definition: as some object oriented languages, e.g. C++, permit the use of multiple inheritance the classes can be organised into a directed acyclic graph instead of trees.

[1] In fact this metric have been included also in the category of coupling metrics. We decided to present it among the complexity metrics, because the viewpoints suggested by its authors emphasize the complexity aspect. Still we consider that RFC could be considered also seen as a complexity metric.

Viewpoints. For this metric the two authors also suggest three viewpoints that should guide its interpretation and use of the results:

1. The deeper the class in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour.
2. Deeper trees constitute greater design complexity for that class, since more methods and classes are involved.
3. The deeper the class is in the hierarchy, the greater the potential reuse of inherited methods.

Concerning the viewpoints defined before, especially the first two, one theoretical assumption that can be made is that well designed OO systems are those structured as forests of classes, rather than as one very large inheritance tree, as suggested by Basili, Briand and Melo in [BASI 95]. That means that a class located deeper in a class inheritance lattice is supposed to be more fault-prone because the class inherits a large number of definitions from its ancestors.

Critics. No critics were found in the researched literature. Instead, Li and Henry analysed this metric together with NOC, that will be presented below, from the perspective of coupling. Their viewpoint is that inheritance is a mechanism by which coupling is realised. They call it coupling through inheritance [LI 93].

20.3.2.2 Number of Children – NOC

Definition 11 *The Number of Children (NOC) represents the number of immediate subclasses subordinated to a class in the class hierarchy.*

Viewpoints

1. The greater the number of children, the greater the reuse, since inheritance is a form of reuse.
2. The greater the number of children, the greater the likelihood of improper abstraction of the parent class. If a class has a large number of children, it may be a case of misuse of subclassing.
3. The number of children gives an idea of the potential influence a class has on the design. If a class has a large number of children, it may require more testing of the methods in that class.

An immediate conclusion is that the classes with a greater number of children have to provide more services in different contexts, and thus they should be more flexible. An assumption that can be made is that such classes will introduce more complexity in the design.

Critics. No critics were found in the researched literature. Instead, Li and Henry analysed this metric together with DIT, that was already presented, from the perspective of coupling.

20.3.3 Metrics on Coupling Between Classes

The relationships between classes represent a major aspect of the object oriented design. Analysing the viewpoints suggested for the different coupling metrics, we can remark that the reuse degree and the maintenance effort for a class are decisively influenced by the coupling level between classes. That means that this kind of metrics should help project-designers to design the classes and especially the relations between them, so that the coupling level might be as low as possible.

20.3.3.1 Coupling Between Objects – CBO

Definition 12 *The Coupling Between Objects (CBO) for a class is a count of other classes to which it is coupled.*

Viewpoints

1. Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse it in another application.
2. In order to improve modularity and promote encapsulation, inter-object class couples should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design and therefore maintenance is more difficult.
3. A measure of coupling is useful to determine how complex the testing of various parts of a design is likely to be. The higher the inter-object class coupling, the more rigorous the testing needs to be.

Critics Although the definition of coupling proposed by Chidamber and Kemerer in [CHID 91] was widely accepted and appreciated in the literature some deficiencies have been identified. We will mention some the most important critics found. In [HITZ 95] the authors make a survey of the previously defined metrics that should reflect the coupling between classes. Their analysis starts exactly from this definition of coupling given by Chidamber and Kemerer.

The first critic of Hitz and Montazeri on this metric is that it does not differentiate between the strength of couples, assuming that all couples are of equal strength. They differentiate and make a hierarchy of coupling strength. The worst coupling that has been identified is the direct access of foreign instance variables. Also the coupling realised by sending messages to a component of one of the object's components is considered stronger (worse) than sending messages to the object itself. Access to instance variables of foreign classes constitutes stronger coupling than access to instance variables of super-classes. Another differentiation can be made between passing a message with a wide parameter interface which yields a stronger coupling compared with a message with a slim interface.

The second critic found in [HITZ 96b] concerns the fact that it is not clear whether messages sent to a part of self - i.e. to a instance variable of class type - contribute to CBO or not. Using strictly the definition given by Chidamber and Kemerer this kind of class coupling should not be considered. This aspect is also analysed by Li and Henry in [LI 93]. They also propose a metric that measures exactly this type of coupling (DAC). We will present this metric below.

The third deficiency of the CBO metric is that it neglects inheritance related connections, that means they exclude from their measure the couples realised by immediate access to instance variables inherited from superclasses, a kind of coupling considered to be among the worst types of coupling.

20.3.3.2 Data Abstraction Coupling – DAC

Li and Henry propose also a set of metrics on coupling [LI 93]. The metrics correspond to the two from the three forms of object coupling that they have identified: coupling through data abstraction, coupling through message passing and coupling through inheritance.

Definition 13 *The metric which measures the coupling complexity caused by ADT's is called Data Abstraction Coupling (DAC) and is defined as:*

DAC = number of ADT's defined in a class

A class can be viewed as an implementation of an abstract data type (ADT). A variable declared within a class X may have a type of ADT which is another class definition, causing a particular type of coupling between the X and the other class, since X can access the properties of the ADT class. This type of coupling may cause a violation of encapsulation if the programming language permits direct access to the private properties in the ADT.

Viewpoints

1. The number of variables having an ADT type may indicate the number of the data structures dependent upon the definitions of other classes. The more ADT's a class has, the more complex is the coupling of that class with other classes.

Critics. No critics were found in the literature for this metric. Instead it is positively cited and accepted in the software metrics literature. For example,Hitz and Montazeri cite it as an alternative to the CBO metric from Chidamber and Kemerer.

20.3.3.3 Message Passing Coupling – MPC

One communication channel the object oriented paradigm allows is message passing. When an object needs some service that the other object provide, messages are sent from the object to the other objects. A message is usually composed of the object-ID, the service (method) requested, and the parameter list for that method. Although messages are passed among objects, the types of messages passed are defined in classes. Therefore message passing is calculated at the class level instead if the object level.

Definition 14 *Message Passing Coupling (MPC) is used to measure the complexity of message passing among classes. Since the pattern of the message is defined by a class and used by objects of the class, the MPC metric is also given as an indication of how many messages are passed among objects of the classes:*

MPC = number of send-statements defined in a class

Viewpoints

1. The number of messages sent out from a class may indicate how dependent the implementation of the local methods are upon the methods in other classes.

Critics. This may not be indicative of the number of messages received by the class. That means that in defining this metric as a coupling measure the authors did not take into account the dependencies of other classes to the class being analysed.

20.3.3.4 Number of Local Methods – NOM

Another metric used is the Number of local Methods (NOM) in a class. Since the local methods in a class constitute the interface increment of the class, NOM serves the best as an interface metric. NOM is easy to collect in most object oriented programming languages.

NOM = number of local methods in a class

Viewpoints

1. The number of methods in a class may indicate the operation property of a class. The more methods a class has, the more complex the class' interface has incremented.

Critics. This metric is just an interface metric. By itself it does not bring very much. In order to be used it has to be combined with the previously presented metric, that is MPC. But in combining the two metrics, we will obtain a new metric that is in fact another way of expressing Chidamber & Kemerer's RFC metric. May be this is one reason for which this two metrics (i.e. MPC and NOM) where not discussed in the literature, as they don't bring anything new.

20.3.3.5 Change Dependency Between Classes – CDBC

In [HITZ 96b] the authors propose two metrics for the measurement of coupling. The first one is called Change Dependency Between Classes (CDBC). In order to give the definition of this metric we need to define some concepts used by the authors in this definition.

Definition 15 *By state of a class we refer to class definition and the program code of its methods, i.e. a version of the class implementation.*

Definition 16 *The Class Level Coupling (CLC) represents the coupling resulting from state dependencies between classes during the development life-cycle.*

The metric refers to the case of class level coupling CLC in the context of unstable server classes. In the sequel, CC will signify the dependent client class and SC the server class being changed.

In order to define the CDBC attribute we have to investigate the possible relationship types. This can be synthesised into table 20.2.

	α = **Number of methods of CC potentially affected by a change**
SC is not used by CC at all	0
SC is the class of an instance variable of CC	n
Local variables of type SC are used within j methods of CC	j
SC is a superclass of CC	n
SC is the parameter type of j methods of CC	j
CC accesses a global variable of class SC	n

Table 20.2: Relationship types between CC and SC and their corresponding contribution α to change dependency.

The maturity of the SC also plays an important role in the definition of the CDBC attribute as the (value is only relevant for those parts of SC that are subject to change. If SC represents a mature abstraction, its interface is assumed to be more stabile than its implementation. Therefore a factor k was introduced $(0 \leq k \leq 1)$ corresponding to the stability of SC's interface. Using the notions introduced until now, the degree of CDBC can be expressed mathematically as follows:

$$A = \sum_{implement=1}^{m1} \alpha_i \; + \; (1-k) \sum_{interface=1}^{m2} \alpha_i$$

where the first sum in the relation represents the sum of (values corresponding to the m1 accesses of CC to the implementation of the SC class; the second sum in the relation represents the sum of α value corresponding to the $m2$ accesses of CC to the interface of the SC. Because the SC class has a stability degree of k this term of the expression is multiplied with $(1-k)$representing the instability degree of the SC class. Knowing A, CDBC value is defined as:

CDBC(CC,SC) = min(n, A)

Viewpoints

1. The Change Dependency Between Classes metric (CDBC) determines the potential amount of follow-up work to be done in a client class CC when the server class SC is being modified in the course of some maintenance activity.

Critics. The CDBC metric was not very much discussed in the literature. In fact this is already a deficiency of the metric. Although it has some interesting aspects it was not validated until now by the specialists.

Another deficiency of this metric is the calculation of the stability degree k of the class. This factor is a very important, but there are no concrete indications given by the authors of this metric on how to compute it; thus it must be determined empirically. Following this conclusion, we can affirm that the stability degree k of a class has to be provided by the designer of the class. This is a big disadvantage in a re-engineering process as in most of the cases those who re-design the system do not know very well the design details and history of that system.

20.3.3.6 Locality of Data – LD

The second metric on coupling is called Locality of Data (LD). This metric represents an attribute directly connected with the quality of the abstraction embodied in the class.

Definition 17 *The Locality of Data (LD) metric is defined by relating the amount of data local to the class to the total amount of data used by the that class. For the C++ language the LD value for a class C can be mathematically expressed by the relation:*

$$LD = \frac{\sum_{i=1}^{n} \mid L_i \mid}{\sum_{i=1}^{n} \mid T_i \mid}$$

with,

$$M_i (1 \leq i \leq n) \textit{ methods of class } C,$$

but excluding all trivialread/write methods for instance variables;

$$T_i (1 \leq i \leq n) \textit{ the set o all variables used in} M_i,$$

except for non-static local variables defined in M_i;

$$L_i (1 \leq i \leq n)$$

the set of class-local variables accessed in M_i, directly or via read/write methods.

The authors include following variables in the category of class-local ones: non-public instance variables of class C, inherited protected instance variables of its superclasses, static variables defined locally in M_i.

We have to make the following observation: for the sake of robustness of the measure all the non-static local variables defined in M_i are not counted, as they don't play a major role in the design.

Viewpoints

1. The Locality of Data metric (LD) influences the class's reuse potential.
2. Classes with high data locality are more self-sufficient than those with low data locality.
3. This metric also influences another external attribute: the testability of the class. The higher the LD value the easier it is to test the class.

We should note that the main point of this metric is the notion of self-sufficiency! A class with a high self-sufficiency, means that the class uses very few data's that are not "local" to it. In fact the "non-local" data that can be accessed by the methods of a class are: global variables and public instance variables of the class, or of the superclasses.

Critics. The LD metric was not very much discussed in the literature. In fact this is already a weakness of the metric. Like the CDBC metrics, although it has some interesting aspects it was not validated until now by the specialists.

20.3.4 The Measurement of Cohesion

Another important aspect in the object oriented design is the level of cohesion for each class. The role of this category of metrics is to detect the classes with a low cohesion level, as these classes represent possible flaws in the design. These classes might be the subject of a redesign process, in most of the cases by splitting. It is mostly desirable to have classes with a high level of cohesion.

20.3.4.1 Lack of Cohesion in Methods – LCOM

Definition 18 *Consider a class C_1 with n methods $M_1, M_2, \ldots, M_n$. Let $\{I_i\}$ represent the set of instance variables used by method M_i. There are n such sets $\{I_1\}, \ldots, \{I_n\}$.*

$$\begin{aligned} \textit{Let } P &= \{(I_i, I_j) \mid I_i \cap I_j = \phi\} \\ \textit{and } Q &= \{(I_i, I_j) \mid I_i \cap I_j \neq \phi\}. \end{aligned}$$

If all n sets$\{I_1\}, \ldots, \{I_n\}$ are ϕ let $P = \phi$. Then:

$$\begin{aligned} LCOM &= \mid P \mid - \mid Q \mid, \textit{ if } \mid P \mid > \mid Q \mid \\ &= 0 \textit{ , otherwise.} \end{aligned}$$

In other words the LCOM value represents the difference between the number of pairs of methods in class C that uses common instance variables and the number of pairs of methods that do not use any common variables.

Viewpoints

1. Cohesiveness of methods within a class is desirable, as it promotes encapsulation.
2. Lack of cohesion implies classes should probably be split into two or more subclasses.
3. Any measure of disparateness of methods helps to identify flaws in the design.
4. Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

Critics. This metric has been criticised by Hitz and Montazeri in [HITZ 96a]. The two authors demonstrate that Chidamber and Kemerer's definition for LCOM does not obtain coherent results for similar cases and exhibits some anomalies regarding the intuitive understanding of the attribute of cohesiveness. Although they agree with the definition of LCOM presented in [LI 93] they propose a graph-theoretic formulation, which they hope to be more precise. Their new definition leads to a second, improved version of LCOM, and subordinated to it, which helps to differentiate among the ties in cases of LCOM = 1. We will present this metric in the section 20.3.4.2 of this section. It also has to be mentioned the fact that Hitz and Montazeri, like some other authors generally criticise the use of a subset from Weyuker's axioms by C&K in order to theoretically validate the definition of their proposed metrics. For more details concerning this point please review the section 20.2.2.2 in section 20.2.

Another critic to this metric comes from Bieman and Kang in [BIEM 95]. According to them, LCOM is effective at identifying the most non-cohesive classes, but it is not effective at distinguish between partially cohesive classes. LCOM indicates lack of cohesion only when, compared pairwise, fewer than half of the paired methods use the same instance variables. In this paper the authors present two cohesion measures that are sensitive to small changes in order to evaluate the relationship between cohesion and reuse. We will present the two alternative metrics proposed by Bieman and Kang in section 20.3.4.3 later in this section.

20.3.4.2 Improvement of LCOM

As already mentioned in the previous section of this section, Hitz and Montazeri propose an improvement of the LCOM metric [HITZ 95].

Definition 19 *Let X denote a class, I_x the set of instance variables for class X, and M_x the set of methods. We will also consider a simple, undirected graph $G_x(V, E)$ with*

$$V = M_x$$

and

$$E = \{(m, n) \in V \times V \mid \exists i \in I_x : (m \textit{ accesses } i) \wedge (n \textit{ accesses } i)\}.$$

That signifies exactly those vertices are connected that in the graph that have at least one common instance variable. We can now define LCOM(X) as the number of connected components of G_x , that is the number of "clusters" operating on disjoint sets of instance variables.

One of the critics on LCOM mentioned in [HITZ 95] was the lack of differentiation between the classes with LCOM = 1. Among these classes there are still more and less cohesive classes possible. Especially for big classes, it might be desirable to refine the measure to tell the structural difference between the members of the set of classes with LCOM = 1. The two extreme cases are this:

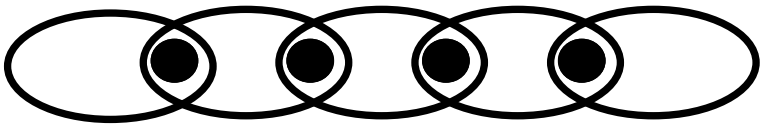


Figure 20.2: Sequential cohesion

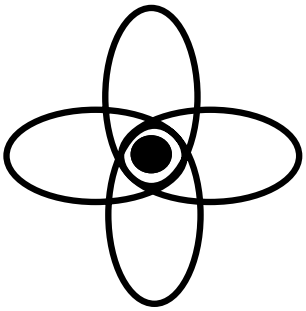


Figure 20.3: Maximal cohesion

- *Sequential cohesion.* This case is described in figure 20.2.

  In this case each method has a set of common instance variable with two other methods. This case corresponds to a minimum cohesive class design with LCOM=1. The cardinality of E for this case is:

  $$\mid E \mid = n - 1$$

- *Maximal cohesion.* This case is described in Figure 2 below:

  In this case all n methods access the same set of instance variables. This case is mapped to the complete graph with the cardinality of E taking the value:

  $$\mid E \mid = \frac{n(n-1)}{2}$$

  Thus we can break many of the ties a in the set of classes yielding LCOM = 1. The more edges in the graph G_x , the higher the cohesion of the class. For convenience we map $\mid E \mid$ into the interval [0,1]:

  $$C = 2\,\frac{\mid E \mid -(n-1)}{(n-1)(n-2)}$$

Discussion and Conclusions. For classes with more than two methods, C can be used to discriminate among those cases with LCOM = 1 as C gives us a measure of the deviation of a given graph from the minimal cohesive case. The two authors also observe that the anomalies they found for LCOM have remained undiscovered in [CHID 94] in part because the validation of the representation condition has been substituted by Weyuker's axiom set. They concluded that any set of validation criteria should only be employed in addition to the more fundamental representation condition.

20.3.4.3 Tight Class Cohesion – TCC

Bieman and Kang propose in [BIEM 95] a set of two metrics on cohesion, that should eliminate the weaknesses that they observed on LCOM.

Definitions and Notations. In order to understand the definition of the Tight Class Cohesion (TCC) metric we have to list some of the concepts the authors use in this article. Class cohesion is defined in terms of relative number of connected methods in the class.The authors identify two mechanisms by which individual methods are tied together:

1. The MIV relations, that involves communication between methods through shared instance variables,
2. Call relations, that involves the sending of messages directly (or indirectly) from one method to another.

A client class can access only visible components of the class. In this model, invisible components of a class are included only indirectly through the visible ones. Therefore, class cohesion is modelled as the MIV relations among all visible methods (not including constructor or destructor functions) in the class.

There are three options to evaluate cohesion of a subclass:

- include all inherited components in the subclass
- include only methods and instance variables defined in the subclass
- include only inherited instance variables.

A method is represented as a set of instance variables directly or indirectly used b the method. The representation of a method is called abstract method AM.

- $DU(M)$ is the set of instance variables directly used by a method M
- $IU(M)$ is the set of instance variables indirectly used by a method M

Abstract Methods. Abstracted Class. Local Abstracted Class.

Definition 20 *A class can be represented as a collection of abstract methods (AM) where each AM corresponds to a visible method in the class. The abstract method set for a method M is the reunion of the set of instance variables directly used by M and the set of instance variables indirectly used by M. Based on the notations we made above, theset of abstract methods for method M can be formally defined using set operators as:*

$$AM(M) = DU(M) \cup IU(M)$$

Definition 21 *The representation of a class using abstract methods is called an abstracted class (AC). An abstracted class corresponding to a class C can be formally expressed as:*

$$AC(C) = \|AM(M) \mid M \in V(C)\|$$

where $V(C)$ is the set of all visible methods in a class C and in the ancestors of C. The AM's of different methods can be identical thus there can be duplicated elements in AC. Therefore, AC is a multi-set (denoted by $\|$ and $\|$)

Definition 22 *A local abstracted class (LAC) is a collection of AM's where each AM corresponds to a visible method defined only within the class:*

$$LAC(C) = \|AM(M) \mid M \in LV(C)\|$$

where $LV(C)$ are the visible methods defined only within the class C

Number of Direct Connections. Number of Indirect Connections. The authors define two measures of class cohesion based on the direct and indirect connections of method pairs. Let $NP(C)$ to be the total number of pairs of abstract methods in $AC(C)$. NP is the maximum possible number of direct or indirect connections in a class. If there are n methods in class C, then:

$$NP(C) = \frac{n(n-1)}{2}$$

Let $NDC(C)$ to be the number of direct connections and $NIC(C)$ be the number of indirect connections in $AC(C)$.

Definition 23 of TCC *Tight class cohesion (TCC) is the relative number of directly connected methods. Formally this can be expressed as:*

$$TCC(C) = \frac{NDC(C)}{NP(C)}$$

Viewpoints

1. TCC indicates the degree of connectivity between visible methods in a class.

20.3.5 Survey of Experimental Result Reports

One of the key problems in the study of metrics is the lack of practical experience in the use object oriented metrics. Searching in the literature we can easily observe that there is a big disproportion between the number of defined metrics and the number of experimental reports. As we will observe from this section a very small number of metrics were experimentally validated. The most of the researchers who have focused their interest on the suite of metrics proposed by Chidamber and Kemerer [CHID 94] as this metrics are the best known metrics so far. Some of the authors - like Bieman & Kang - have added to the theoretical definition of the new metrics they were proposing, the results of their experimental studies regarding their metrics. We will take a survey of the most important such experimental results found in literature.

20.3.5.1 Chidamber&Kemerer's Experimental Results

In the same article [CHID 94] where the two authors defined a suite of metrics for the object oriented design, we find also an experimental result report for each of the six metrics that they defined.

Systems Description They used for research two systems that were strongly differing from each other. The first one (called Site A) is a collection of C++ class libraries. The metrics data consist of 634 classes from two C++ class libraries that are used in the design of graphical user interface (GUI). These libraries were typically used in conjunction with other C++ libraries and traditional C-language programs in the development of software used to UNIX workstation users. The second system (called Site B), came from a semiconductor manufacturer and uses the Smalltalk programming language for developing flexible machine control and manufacturing systems. Metrics were collected on the class libraries used in the implementation of a CAM-system for the production of VLSI circuits. The metrics data were collected from 1459 classes, designed by over 30 engineers, that had an extensive training and experience with object orientation.

Analysis of the Results

- WMC Metric - The most interesting aspect was at this metric that despite the differences between Site A and B, the distribution of the values at the two sites where very similar. This seemed to suggest that the most classes tend to have a small number of methods (0 to 10), while a few outliers tend to have a big number of them (106 - Site A; 346 - Site B).

- Another observation that was made is that an increased attention in the testing phase should be accorded to the outliers that have also a big number of children, as through their reuse potential their methods may affect many other classes.

- DIT Metric - Both Site A and Site B libraries have a low median value for this metric (1 at Site A; 3 at Site B). This suggests that most classes in an application tend to be close to the root in the inheritance hierarchy.

- At both sites, the libraries appeared to be top-heavy suggesting that designers may not be taking advantage of reusing methods through inheritance. The Smalltalk application has a higher depth of inheritance due, in part, to the library of reusable classes that are a part of the language.

- Another interesting aspect is the maximum value of DIT is rather small (10 or less). One possible explanation is that designers tend to keep the number of levels of abstraction to a manageable number in order to facilitate comprehensibility of the overall architecture of the system. Designers may forsake reusability through inheritance for simplicity of understanding.

- NOC Metric - Like the WMC metric, an interesting aspect of NOC data is the similarity of the distribution of the metric values at Sites A and B. This seems to suggest that classes in general have a few immediate children and that only a very small number of outliers have many immediate subclasses. This further suggests that designers may not be using inheritance of methods for designing classes. The majority of the classes (78% at Site A; 68% at Site B) have no children.

- One explanation for the small NOC count could be that the design practice followed at the two sites dictated the use of shallow inheritance hierarchies[2] A different explanation could be the lack of communication between different class designers and therefore that reuse opportunities are not being realised.

- The outlier at Site A is a class with 42 subclasses, that implemented a GUI-command. Further, none of these subclasses had any subclasses of their own.

- CBO Metric - Both Site A and B class libraries have skewed distributions for CBO, but the Smalltalk application has a relatively high median values.

- One explanation is that the difference of type of application are responsible for the difference. Another explanation may be the difference between the programming languages. The C++ programs tend to have smaller CBO values. A third explanation might be that the coupling between classes is an increasing function of the numbers of classes.

- The low median values (0 at Site A; 9 at Site B) suggest that at least 50% of the classes are self-contained and do not refer to other classes. Since a fair number of classes at both sites have no parents or no children, the limited use of inheritance may be also responsible for the small CBO value.

- Analysing the outliers at Site B revealed that the classes that are responsible for managing interfaces have high CBO values.

[2] Some C++ designers systematically avoid subclassing in order to maximize operational performance.

- RFC Metric - The data from both Site A and Site B suggest that most classes tend to be able to invoke a small number of methods, while a few outliers may be most profligate in their potential invocation of methods. This reinforces the argument that a small number of classes may be responsible for a large number of the methods that executed in an application, either because they contain many methods (case of Site A) or that they call many methods.
- Another interesting aspect is the difference in values for RFC between Site A and Site B. The median and the maximum values are higher than the RFC values at Site A. This may relate to the complete adherence to object oriented principles in Smalltalk which necessitates extensive method invocation, whereas C++'s incremental approach to object orientation gives designers alternatives to message passing through method invocation.
- LCOM Metric - At both sites, LCOM median values are extremely low, indicating that at least 50% of classes have cohesive methods. In other words, instance variables seem to be operated on more than one method defined in the class. The Site A application has a few outliers that have low cohesion, as evidenced by the maximum value 200. In comparison, the Site B application has almost no outliers, which is demonstrated by the difference in the shape of the distributions.
- A high LCOM value indicates disparateness in the functionality provided by the class. The metrics can be used to identify classes that are attempting to achieve many different objectives, and consequently are likely to behave in less predictable ways than classes that have lower LCOM values. Such classes could be more error prone and more difficult to test and could possibly be disaggregated into two or more classes that are more well defined in their behaviour.

20.3.5.2 Experimental Study on C&K Metrics Suite

In the article "A Validation of Object-Oriented Design Metrics as Quality Indicators", the three authors, Basili, Briand and Melo, present the results of a study they conducted September-December 1994 at the University of Maryland, in which experimentally investigated Chidamber&Kemerer's suite of metrics from the perspective of fault-proness for a class [BASI 95].

System Description The system used for research was a medium-sized management information system that supports the rental/return process of a hypothetical video rental business, and maintains customer and video data bases. The same system was developed at the same time, by eight student teams in the C++ language. The students were not required to have previous experience or training in the application domain or OO methods. The development process was performed according to a sequential software-engineering life-cycle model derived from the Waterfall model. Errors found in the analysis and design phase were reported to the students in order to maximise the chances that the implementation began with a correct OO analysis and design. The testing phase was accomplished by an independent group composed of experienced software professionals.Each team was provided with following libraries: MotifApp - for the manipulation of a GUI interface; GNU library - for the manipulation of string, files, lists; and C++ database library that provided a C++ implementation of multi-indexed B-Trees. Together with the libraries they got program samples and complete documentation.

Data Collection The source code of the C++ programs were collected at the end of the implementation phase. Also data about these programs and data about errors found during the testing phase and fixes during the repair phase were also collected.

Analysis of the Results The analysis methodology that they used in order to detect the relationship between metrics and the fault-proneness of classes was the logistic regression. Logistic regression is a classification technique based on the maximum likelihood estimation [HOSM 89]. In this case, a careful outliers analysis must be performed in order to make sure that the observed trend is nit the result of few

observations. The analysis consisted of two steps: Firstly, the univariate logistic regression was used to evaluate the relationship of the metrics in isolation and fault-proneness. Secondly, a multivariate logistic regression was performed, to evaluate the predictive capability of those metrics that had been assessed sufficiently significant in the univariate analysis.

- Weighted Methods Count (WMC) - This metric was shown somewhat significant (p = 0.06) overall. For new and extensively modified classes and for UI (Graphical and User Interface) classes, the results are much better: p = 0.0003 an p = 0.0001, respectively. As expected the larger the WMC, the larger the possibility of fault detection. The authors suggest that these results can be explained by the fact that the internal complexity does not have a strong impact if the class is reused verbatim or with very slight modifications. In that case, the class interface properties will have the most significant impact.
- Depth of Inheritance Tree (DIT) - This metric was shown to be very significant (p = 0.0000). As expected, the larger the DIT, the larger the possibility of defect detection. Again, results improve when only new and extensively modified classes are considered.
- Number of Children (NOC) - This metric appeared to be very significant, except in the case of UI classes, but the observed trend is contrary to what was expected. The larger the NOC, the lower the possibility of defect detection. This surprising trend can be explained by the combined facts that the most classes do not have more than one child and that verbatim reused classes are somewhat associated with a large NOC. Since it was observed that reuse was a significant factor in fault density [BASI 95], this explains why large NOC classes are less fault-prone. Moreover, there is some instability across class subsets with respect to the impact of NOC on the probability of detecting a fault in a class. This may be explained in part by the lack of variability on this measurement scale.
- Response for a Class (RFC) - This metric was shown to be very significant overall (p = 0.0000). Predictably, the larger the RFC, the larger the probability of defect detection. However, the logistic R2 improved significantly for new and extensively modified classes and UI classes - from 0.006 to 0.24 and 0.36 respectively. Reasons are believed to be the same as for WMC for extensively modified classes. In addition UI classes show a distribution which is significantly different from that of DB classes: the mean and the median are significantly higher.
- Lack of Cohesion on Methods (LCOM) - This metric was shown to be insignificant and this should be expected since the distribution of LCOM shocks a lack of variability and a few very large outliers. This comes in part from the definition of LCOM where the metrics is set to 0 when the number of class pairs sharing variable instances is larger than of the ones not sharing any instances. This definition is not appropriate as it sets cohesion 0 for classes with very different cohesion and keeps us from analysing the actual impact of cohesion.
- Coupling Between Objects (CBO) -This metric is significant and more particularly for UI classes (p = 0.0000 and R2 = 0.17). But no satisfactory explanation could be found for the differences in pattern between UI and DB classes.

Another conclusions of the authors is that all metrics besides NOC, have a very stable impact across various categories of classes (i.e. DB, UI, New-Ext, etc.). This is somewhat encouraging since it tells us that, in that respect, the various types of components are comparable.

20.3.5.3 Bieman&Kang's Experiments with TCC and LCC

System Description The system used for the experimental study was a C++ system having 25.000 non-commented lines of code. In order to make the results be more relevant they respected following conditions:

- The classes without methods were not included

- The virtual methods with empty bodies were not included
- Only local cohesion was used to remove the effects of the superclass

Analysis of the Results The conclusions of their experiments are:

- TCC and LCC measures can be used to locate the classes that may have been designed inappropriately

Considering two forms of class reuse: (a) instantiation reuse - that is measured as the number of classes where the class is instantiated; (b) inheritance reuse - that is the number of classes which inherit the class (the number of direct or indirect descendants), the conclusions are:

- No relationship was found between class cohesion and instantiation reuse.
- The classes that are reused more frequently exhibit lower cohesion and this relationship holds generally for all levels of depth in the inheritance hierarchy.

20.4 An Experimental Study

In the previous section we presented an overview of some of the important object oriented metrics found in the literature. In this section we will cover an experimental study that we conducted based on the metrics from the literature. The section is composed of two parts: in the first part we will describe the process of selecting the metrics to be studied from all the metrics presented before, including criteria that were used for the selection; in the second part we will describe the results and the conclusions of the experimental study.

20.4.1 Metrics Selection

In the previous section we presented and discussed a number of metrics that we found in the literature. From this set of metrics we selected a reduced set of eight metrics that we want to study more closely. The selection of the metrics is the first step of our experimental study. It is based on the literature research that we made, and it conforms to the selection criteria that we will mention below. Corresponding to the figure that we used before, we are analysing in this section the first step: the selection of the metrics.

The Selection Criteria In order to make a selection, we first need to establish a set of criteria that should guide the selection process. The establishment of these criteria has to consider the ultimate goal of the study. As our goal is to find out which of the metrics can be successfully used in order to improve the quality of a project design, there will be two criteria: the theoretical evaluation of the definition of the metric and the conclusions of some, very few reports on the experimental use of the metric.

- *Critical Evaluation from the Literature.* The most of the metrics defined in the literature are the subject of much debate among the specialists. Every time a new metric is defined it provokes a lot different reactions. Thus the first selection criteria is the way the metric is perceived by the specialists. This is why as we discussed the different metrics we have also presented some of the important critics that were found. If a particular metric definition was strongly criticised in the literature it means that it has small chances to be a good indicator of the design quality. On the other hand a metric that encountered a positive reaction from the researchers, has better chances to be a useful one.

- *Results and Conclusions of Previous Experimental Reports* This criteria is related to the conclusion of the previous experimental works concerning this metrics. There are not very many such reports, and this is also a reason for making this study. But still there are, as we have already shown in the last section some such studies where some of the most important metrics are analysed. We used that results to eliminate some of the metrics that were proved not to be relevant, as we will see in the next section.

20.4.2 The Set of Selected Metrics

We will discuss the selection of the metrics described in section 20.3, by grouping them in the same manner they where grouped before, that means classified conforming to the category to which they theoretically belong. Finally we will synthesise the results of the selection in a table.

Class Complexity Metrics. Analysing the several viewpoints suggested for each metric, it becomes clear that we generally expect from these metrics to be predictors of the maintenance effort for a class. They may also be an indicator of the central classes in a project. We may also say that the place of the complexity metrics in the object oriented design process may be the phase of identifying the semantics of classes, and some of the metrics may be used also in the class identification phase.

- **Weighted Method Count – WMC**. We decided to select this metric from Chidamber & Kemerer's suite because it was respecting the two criteria that we established. On the one hand there were no serious critics concerning the definition of the metrics; on the other hand the conclusion from the two experimental reports (i.e. C&K's report and the report from Basili, Briand & Melo) is that the metric is really giving indications that conform to the theoretical assumed viewpoints.
- **Response For a Class – RFC**. We decided to select this metric from Chidamber & Kemerer's suite because it was respecting the two criteria that we established. On the one hand, the only critic that we found was that it includes both aspects of class complexity and aspects of coupling measure, but we considered that it is interesting to test experimentally if in the practical use of this metrics this critic is a real disadvantage. We also decided to include this metric in the set of metrics to be studied as two experimental reports that we found (i.e. those mentioned at WMC) is that the metric is really giving indications that conform to the theoretical assumed viewpoints.
- *SIZE1 and SIZE2*. We decided not to select this metric from Li & Henry because it presented severe weaknesses in respect to the two criteria that we established. Firstly, the SIZE1 metric is defined for the traditional approach and, as the author himself underlined the size factor in an object oriented program has not been well established. Also, these metrics were not discussed or experimentally studied in other papers and even the authors have concluded that the two metrics alone do not give useful information in predicting maintenance efforts.

Class Hierarchy Layout Metrics It is clear that what we expect from this kind of metrics is that they should be very important indicators in the early phases of the design, i.e. in the classes identification phase, where the possible classes of the project are being identified. From this perspective we analysed these category of metrics.

- **Depth of Inheritance Tree – DIT**. We decided to select this metric from C& K's suite because it was respecting the both selection criteria that were required. On the one hand there were absolutely no critics concerning the definition of the metrics, and more this metrics together with NOC were widely discussed and appreciated; Also the conclusion from the two experimental reports (i.e. those mentioned at WMC) is that the metric is really giving indications that conform to the theoretical assumed viewpoints.

- **Number of Children – NOC**. We decided to select this metric from Chidamber & Kemerer's suite because it was respecting the two criteria that we established. In the same manner as it was in the case of DIT no absolutely no critics concerning the definition of the metrics, and more this metrics together with NOC were widely discussed and appreciated; Also the conclusion from the two experimental reports (i.e. those mentioned at WMC) is that the metric is really giving indications that conform to the theoretical assumed viewpoints.

Coupling Metrics The relationships between classes represent a major aspect of the object oriented design. What we suppose that this kind of metrics should do is to help project-designers to design the classes and especially the relations between them, in order to lower the coupling level, as much as it is possible.

- *Coupling Between Objects – CBO*. We decided not to select this metric from C&K as it is very criticised in the literature. The critics that were made posses strong arguments as we have seen this in the previous section. More than that the researchers that criticised it proposed alternative and more comprehensive definitions for coupling metrics (e.g. LD, DAC) The conclusion from the two experimental reports (i.e. C&K's report and the report from Basili, Briand & Melo) were not negative at all. Basili & co. appreciated that the metric is relevant but, some of the results can not be explained. The decision not to include this metric was based on the fact that, we posses better definitions for coupling.
- **Data Abstraction Coupling – DAC**. We decided to select this metric from Li and Henry because it is often cited in the literature and seen as an partial alternative to C&K's CBO metric covering the coupling through ADT's [HITZ 96b]. On the other hand this metrics was studied experimentally very little, except by the authors themselves. As the results of their study was positive we still decided to select it.
- *Message Passing Coupling – MPC*. We decided not to select this metrics proposed by Li & Henry because it presents severe weaknesses in respect with the two criteria that we defined. Firstly, the MPC metric is not taking in account the dependencies of Also, the metrics was not discussed or experimentally studied in other papers and even the authors have concluded that the metric alone does not give useful information in predicting maintenance efforts.
- *Locality of Data – LD*. We decided not to select this metric from Hitz & Montazeri because from the theoretical point of view there are some unclear aspects of its definition, although it is covering some aspects of coupling that were never before analysed. The only problem with this metric is that until now it was not validated and that there is no experimental report.
- **Change Dependency Between Classes – CDBC**. We decided to select this metric from Hitz&Montazeri first, because it is a metrics that directly addresses an important aspect in re-engineering: maintenance effort. A second reason for selecting this metric refers to the very interesting aspects included in its definitions, like considering the stability of the server class and calculating the metric for a *pair* of classes instead of computing it for only one class.

Cohesion Measurement Metrics Another important aspect in the object oriented design is the level of cohesion for each class. The classes with a low cohesion level represent possible flaws in the design, and we hope that by using cohesion metrics we will be able to detect them. These classes might be the subject of a redesign process, in most of the cases by splitting.

- *Lack of Cohesion on Methods – LCOM*. We decided not to select this metric from C&K as it is the most criticised metric from the whole suite in the literature. The critics have a well argumented justification and we referred to this in the previous section. Also the conclusion from the two experimental reports (i.e. C&K's report and the report from Basili, Briand & Melo) are not positive at all.

Basili & co. appreciated in their report that the indications that they got from this metric were totally irrelevant.

- **Tight Class Cohesion – TCC**. We decided to select this metric proposed by Bieman and Kang because it was fully respecting the two criteria that we established. The most important aspect is that it proved - from the experimental research of its authors - to be a very efficient option for cohesion measurements.

Conclusions on the Metrics Selection. We will conclude this section presenting in a synthetically form the results of the metrics selection stage:

Name of the metric	Selected?
Weighted Method Count – WMC	YES
Response For a Class – RFC	YES
SIZE1 and SIZE2	NO
Depth of Inheritance Tree – DIT	YES
Number of Children – NOC	YES
Coupling Between Objects – CBO	NO
Data Abstraction Coupling – DAC	YES
Message Passing Coupling – MPC	NO
Number of Local Methods – NOM	NO
Locality of Data – LD	NO
Change Dependency Between Classes – CDBC	YES
Lack of Cohesion on Methods – LCOM	NO
Tight Class Cohesion – TCC	YES

Table 20.3: A summary of the metrics and their selection status

¿From Table 20.3 we can see that from the initial set of 13 metrics grouped in 4 categories, we decided to continue our study with a number of 7 metrics, distributed like this on the four groups: WMC and RFC are the two metrics selected from the group of class complexity metrics; both NOC and DIT were selected from the class hierarchy layout group; DAC and CDBC will be the metrics to be analysed on the aspect of coupling, while TCC will be representing the metrics that are measuring cohesion.

20.4.3 The Experimental Context

In this study we used three different that sites from two organisations. We will refer them in the next sections as Site A, Site B and Site C. Site B and C are two software components that are independent parts of a bigger project developed by the company. The three systems are developed using different C++ environments and were developed for different operating systems. We chose them this way in order to assure the conclusions of this study possess a high degree of generality.

Site A. This site is an object oriented system produced under the Windows NT(operating system, and developed in the Microsoft Visual C++ programming environment, in conjunction with the Microsoft Foundation Classes(library (MFC). The system has 232 classes, and can be therefore considered a large-scale object oriented system. As it is developed more recently than the other two sites, and because it is developed using MSVC it uses many of the advanced C++ facilities (like templates) pure virtual classes that the other two sites are not using. This

Site B and Site C. These two sites represent two object oriented systems produced under the SunOS(operating system, and developed using an integrated C++ programming environment based on the GNU C/C++ compiler, and of course it is using a number of C++ and C libraries. Both software components, that are part of a very large user interface(UI) project. The company that produced this software is one of the largest companies in the world in the area of communications, and the large application we are speaking about is also in this domain of telecom networks. The large application was split in a number of parts, called subproducts. The component that are including the two sites was intended to offer basic networking service to other subproducts. There are three such subproducts: the switch management system (NSS); the base station management system and the transmission management. The major requirement of this UI subproduct was to assure concrete flexibility at the top-level UI. The top-level UI provides a graphical representation of the of the managed telecom network Thus it is a key component of the whole application.

In order to have an image of the dimensions of the two Sites, we may say that Site B is a quite small component, including just 20 classes, while Site C is a medium one with 63 classes.

We took the decision to use these sites as they are from many points of view different and thus representative for different types of object oriented applications.

20.4.4 The Experimental Results

20.4.4.1 Weighted Method Count – WMC

The Results The metric was studied using two definitions of the complexity function:

- Unitary Complexity - that is to consider that each of the methods of the class has the same complexity, in other words to ignore the complexity of methods.
- McCabe's Cyclomatic Complexity - that means that for each method its complexity has to be first calculated. The McCabe's Cyclomatic Complexity is defined as the number of linearly independent paths in a method and therefore the minimum number of the paths that should be tested.

The results for this metric are presented in Table 20.4.

Complexity Definition	Site Name	Minimum	Average	Maximum
McCabe	Site A	0	21	222
	Site B	0	21	99
	Site C	0	56	393
Unitary	Site A	0	13	113
	Site B	0	10	34
	Site C	0	16	87

Table 20.4: Summary Statistics for the WMC metric

Results Interpretation

- The metric should show us the classes which have the highest degree of complexity, that are the classes who needs more maintenance effort. Our attention has to be focused on these outliers.
- The classes with a high WMC are supposed to be the central classes in a project, as they are contain a high amount of complexity.
- We also expect that this classes should be deep in the class hierarchy, and have few or no children.

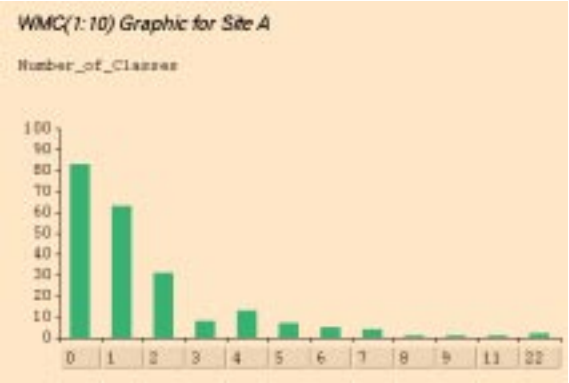


Figure 20.4: WMC Graphic for Site A

Experimental Observations

1. A general observation is that for all sites the most of the classes are simple, and the number of the outliers are few, but having a very high values.
2. The top of the outliers is in some situations (e.g. Site C) is strongly determined by the way we choose the definition of complexity. For example a class can have a large number of very simple methods, and if calculate the complexity using the unitary definition we may obtain completely different results as for the case of using McCabe's cyclomatic.
3. Related with the previous observation, we would recommend not to use the unitary complexity definition, but instead use McCabe definition.
4. An also important conclusion is that in all of the cases the outliers proved to be the central classes of the application. This information can be used at the beginning of a re-engineering (re-designing) process for a unfamiliar software, as it is recommended to start the study of a software with the main classes. We may say that WMC gives very important help in this area.

20.4.4.2 Response for a Class – RFC

The Results. The results for the RFC metric are presented in Table 20.5.

Site Name	Minimum	Average	Maximum
Site A	0	14.66	133
Site B	0	10.25	49
Site C	0	31.63	155

Table 20.5: Summary Statistics for the RFC metric

Results Interpretation.

- The RFC value indicates us the measure of its protocol it offers to other classes and the number of methods it invokes from other classes.
- We also observe from its definition, that this metric refers to two aspects: coupling and complexity. It is interesting to see the way it practically is related to WMC in the aspect of complexity and to DAC in the aspect of coupling.

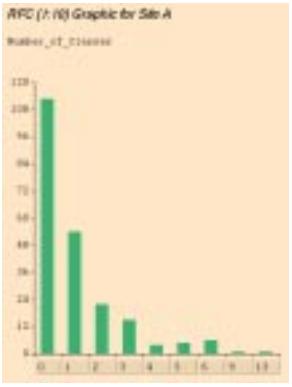


Figure 20.5: RFC Graphic for Site A

Experimental Observations

1. Comparing the outliers from WMC with those from RFC, in Site A the strongest outlier is same for both metrics, but the other two are different. In Site B the first two outliers are identical; and in Site C the top outlier for WMC is the third outlier for RFC. Another connection observed between the two metrics can be seen below at point 3.

2. In order to obtain comparable results on different projects it would be interesting to normalise the metric by dividing its value to the total number of methods in the class. This will be done in a future research.

3. Another observation that should be made is the big difference of the medium (average) values of Site C compared wit the other two sites, and especially to Site A. Although the Site A is much larger than Site C, the average RFC value for Site C is much higher than in the other Site - 31.63 compared to 14.66(A) and 10.25(B). The same observation could be made also on WMC where the average value is two times higher when using McCabe's definition of complexity. This might indicate that the classes, respectively the methods in Site C are very complex, and that may indicate a high complexity degree and thus a hugh fault-prones rate.

4. We should also remark for this metric the difference in the form of the graphics, for Site B compare with the other two. This site is the smallest one. It seems that smaller applications tend to have their methods better distributed among the classes than larger applications who tend to make a small number of classes be responsible for a large number of methods executed in the program.

20.4.4.3 Number of Children – NOC

The Results. The results for the NOC metric are presented in Table 20.6.

Site Name	Minimum	Average	Maximum
Site A	0	0.38	6
Site B	0	0.10	2
Site C	0	0.11	7

Table 20.6: Summary Statistics for the NOC metric

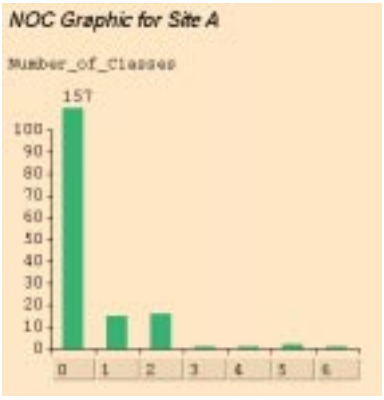


Figure 20.6: NOC Graphic for Site A

Results Interpretation.

- The metrics could tell us how much the reuse facility of the object-oriented approach is used by the designers. If a class has a very low NOC value that means that the class is not very much reused.
- On the other hand extreme high values for NOC may indicate us that the class that has that high value may be an improper abstraction, and that subclassing was misused.
- Also a class with a high NOC value requires more testing effort as it has a higher potential influence in that class.

Experimental Observations

1. All the three sites present the same aspect: the most of the classes tend to have no children (67% in Site A; 95% Site B and 98% Site C) or very few children. It can be observed that the producer of Site B and C is obviously not using the facility of subclassing almost at all. This may be a design practice or a sign of lack of communication among the programmers who implemented this.
2. On the other hand the outlier of Site C is extremely high if we relative to the other classes. That class should be carefully analysed in order to see if there is no possibility of reducing its NOC value. The same thing is valid for all the sites, and may be a practical advice for project managers.
3. A conclusion and in the same time an indication is that classes with a high WMC and in the same time high NOC value are not at all desirable in a project. We suggest as a future work to study more closely the possibility to detect critical classes in a project by analysing a combinations of the two metrics.
4. We also may observe and indicate that the systematic use of this metric during the design phase of a project may help to restructure the class hierarchy and to exploit the common characteristics especially in GUI applications as it is in our case.

20.4.4.4 Depth of Inheritance Tree – DIT

Results. The results obtained for the DIT metric on the three case-studies are presented in Table 20.7.

Site Name	Minimum	Average	Maximum
Site A	0	1.61	9
Site B	0	0.25	1
Site C	0	0.54	1

Table 20.7: Summary Statistics for the DIT metric

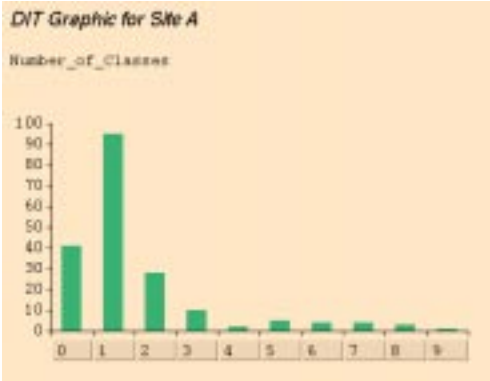


Figure 20.7: DIT Graphic for Site A

Results Interpretation.

- A class with a high DIT value is a class that is situated deep in the class hierarchy, and thus its behaviour is more unpredictable as it may inherit attribute (i.e. methods) from a larger number of classes situated higher in the hierarchy.
- A high DIT for a class has also a positive interpretation, that is: such a class is supposed to take advantage of the reuse possibility.

Experimental Observations

1. The average value is very small, although in Site A is 3-5 times higher than in Site B - Site C. But still values like 1.6 ; 0.25 ; 0.54 are very low, suggesting that generally the most of the classes (100% in Site B and Site C; and 58% in Site A have a DIT ¡ 2) in an application tend to be closer to the top of the hierarchy.
2. It is only Site C which has the majority of its classes having a non-zero DIT value, that means that more than 50% of the classes in that project are derived from another class. This may be a good sign in the sense of reuse, but on the other hand we have to see that the whole application is organised only on two levels, while Site A, although the majority of its classes are at the top of the hierarchy, has a maximum DIT value of 9. We may conclude by saying that the quality of the hierarchy layout has to be determined based not only on all the collected data.
3. We may also conclude that there has to be a trade-off between maintenance requirements and adding more design complexity to a class. Resources between design and testing could be adjusted accordingly.

20.4.4.5 Data Abstraction Coupling – DAC

Results The results obtained for the DAC metric on the three case-studies are presented in Table 20.8.

Site Name	Minimum	Average	Maximum
Site A	0	6.56	43
Site B	0	6.35	21
Site C	0	13.27	53

Table 20.8: Summary Statistics for the DAC metric

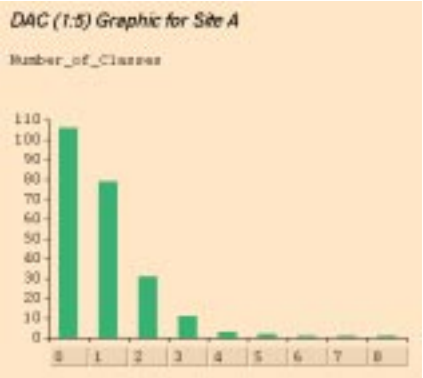


Figure 20.8: DAC Graphic for Site A

Results Interpretation.

- The metric measures the relation of a class with the rest of the system it belongs to. The DAC value for a class is defined as the number of ADT – Abstract Data Types – that are instantiated and used in that class. That means it is desirable that classes have a low DAC value as this indicate that such classes are more self sufficient, and thus less exposed to modifications as the system, i.e. the other classes from the system suffer changes. Based on the previous point we may say that classes with a high DAC value are more unstable, while the other ore more stable when a change occurs in the system.

Experimental Observations

1. We observed as we did also for WMC and RFC, that the Site C has a very high average value compared with the other two sites which have very close values. This may consolidate our assumption that this Site C has a quite poor design and should be reviewed

Quality Improvement MethodBased on DAC metric. We may suggest the following two-step methodology of improving the quality of a projects (a software system) using this metric:

Step I: Identify all the classes with a high DAC value.

Step II: Analyse all the ADT's in that class, by asking two questions:

1. Is it necessary or does it make any sense to declare this complex variable in this class?
2. The method(s) which is using that ADT , shouldn't it belong to another class?

Improvements of the Definition of the DAC Metric.

1. We consider the metric could be more efficient in detecting the coupling in a system if we wouldn't consider in the DAC's definition all the complex variables from a class, but only the user-defined ones. This proposal is based on the observation that the library classes are more stable than the user-defined ones, and thus the coupling to such classes is insignificant compared to the coupling to a user-defined class. What makes coupling to be a problem is the instability of the component to which a class is coupled.

2. The second proposal refers to the fact that the metric is not normalised. This is one of the further study directions that we want to suggest. For the particular case of DAC we propose the following normalisation of the metric: the DAC value for a class should be divided by the total number of classes minus one.

20.4.4.6 The *CDBC* Metric

The Results. Below we present a summary of the results on this metric (Table 20.9):

	Site Name	$k = 0.0$	$k = 0.5$	$K = 1.0$
Minimum	Site A	1	0.5	0
	Site B	1	0.5	0
	Site C	1	0.5	0
Average	Site A	8.64	5.15	1.24
	Site B	8.11	4.77	1.18
	Site C	6.97	3.81	0.39
Maximum	Site A	113	113	113
	Site B	95	48	31
	Site C	189	95	30

Table 20.9: Summary Statistics for the CDBC metric

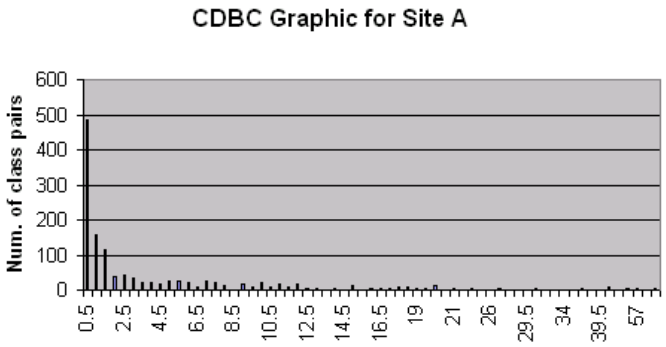


Figure 20.9: CDBC Graphic for Site A

Results Interpretation.

- The metric should measures the potential amount of follow-up work to be done on a client class when the server class is modified in the course of some maintenance activities.
- The follow-up work is expressed by the number of methods from the client class that have to be modified when a change occurs in the server class.
- The metric should measure the strength of the coupling between two classes, that means CDBC can give indications concerning coupling.

Experimental Observations

1. Depending on the *CDBC* value for a couple, we may say that some couples are *strong* or *weak*. We have empirically defined the strong couples having $CDBC < 5$, and we counted the number of strong couples for each client class. The most complex class in Site A was strongly coupled to 15 classes. Comparing this top with the results of the *MIC metric* we concluded the the *order* of the mostly coupled classes remains approximately the same.
2. In using this metric a key point is choosing the value of the stability factor k. We made measurements varying the stability factor ($k = 0, 0.5, 1.0$) and observed the evolution of the top 10 outliers. For $k = 0$ (high instability of the server class) the top client classes were mostly the big classes in the projects. When k was considered at a middle value (0.5) although the top 10 didn't change very much compared to $k = 0.0$, yet a couple of new pairs began to "raise up" in the top. When taking $k = 1.0$ these new pairs took first places in the top.
3. In addition to the previous observation, we remarked that on the one hand there are minor differences between the top 10 for $k = 0$ and $k = 0.5$, but there are major differences between the the top 10 for $k = 0$ and $k = 1.0$.
4. We have analysed the cases where *all* the methods of the client class are potentially affected by changes in the server class (*CDBC* = number of methods in the CC). Our goal in doing this was to detect the most frequent causes for this undesired situation. Two major causes where detected:
   - **Inheritance**. The client class accesses members inherited from one of its superclasses. We observed that for classes having a deeper place in the class hierarchy, this kind of coupling was the strongest and the most frequent one.
   - **Access to Implementation**. The client accesses the implementation of a class, through an instance variable. In most of the cases, it accesses the data fields of a data structure, causing an access to the implementation of the class that encapsulates the data structure.

Conclusions on CDBC

1. The *CDBC* metric proved to be a very good indicator of the *client classes* with a high degree of internal coupling. From this point of view the metrics may be used in the design practice in order to try reducing the strength of coupling, and thus the sensibility towards changes in the system.
2. Another very important and interesting conclusion is that this metric can prove to be useful also from the perspective of the *server class*. In order to reach the first conclusion we grouped the results by the client class. Now, if we group the results by the server class, we can detect the most used classes. It is desirable that such classes should have a high stability, because of their strong influence on the other classes in the system.
3. In other words, two rules have to be obeyed in order to reduce the *CDBC* value:
   - Reduce the number of the accesses of the client class to the implementation of the server class
   - Raise the stability degree of the server classes that are much used.

20.4.4.7 Tight Class Cohesion – TCC

The Results. Below we present a summary of the results on this metric (Table 20.10):

Site Name	Minimum	Average	Maximum
Site A	0	0.43	1
Site B	0	0.42	1
Site C	0.1	0.46	1

Table 20.10: Summary Statistics for the TCC metric

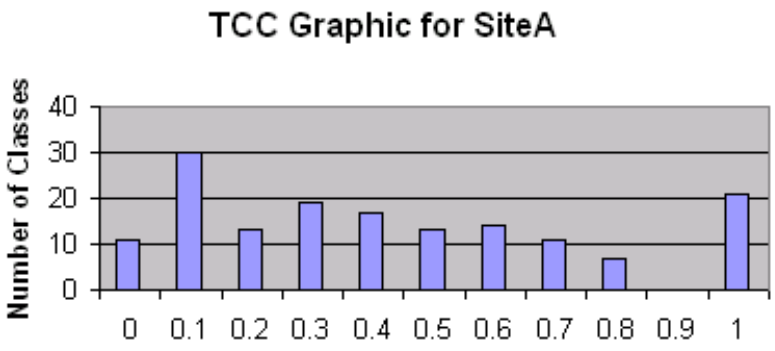


Figure 20.10: TCC Graphic for Site A

Results Interpretation.

- The metric is measuring the internal coupling of a class, i.e. the class cohesion. The TCC value is defined as the fraction from the number of visible methods pairs in a class, that are connected(related) by an access of a common instance variable.
- The values of TCC are normalised in the range between [0,1]. The higher the value of TCC for a class, the stronger the cohesion of that class, and consequently the better the design. Classes that have a TCC-value lower than 0.5 (or better lets say 0.3) are candidate classes for a redesigning process. The redesigning consists in a possible splitting of the class in two or more smaller classes that are consequently more cohesive.
- From another perspective the subjects with a low LCC may indicate classes that have more than one functionality.

Experimental Observations

1. The metric was proved to be very effective for the classes built on the ”classic” principle in object-orientation, that is a class centered on a data-structures, and where methods are manipulating and modify these data structures. These classes are in most of the cases container classes, classes that have been created using design tools like OMT.
2. On the other hand this metric has proved to be not very efficient when classes that are using - mostly by means of inheritance - elements from predefined libraries (like MFC), because in this case many of the methods are not coupled through data structures; the methods are more semantically related.

3. As the authors of the metrics also suggested in applying the metric the abstract methods should not be considered, as abstract methods do not access no instance variables at all.

4. Classes with a big number of methods have in 99% of the cases a low TCC cohesion value. But the reciprocal is not true in the same measure, that means that we found in Sites B, and Site A classes with TCC = 0 and having just a reduced number of methods (8 or less)

5. All the three analysed sites although they have a very different number of classes and are implementing different types of applications have very similar results. The average value is very close to 0.5, although it is under that value.

6. We also found a container class with a very low TCC value. The reason was that the specific type of implementation of that classes. Any container class needs functions acting on two levels: a set of low-level functions that manipulate directly the data-structure (list, table, etc.) and a set of high-level methods, that do not act directly on the data structure but by invoking the low-level functions. In our particular case the designer decided to include all the methods in one single class. If we would have split the class in two classes, one for the data-structure and the low-level methods and another one that defines an instance variable of the low-level class, and implements the container class, than both classes are expected to have high TCC values.

7. The advantages of splitting classes that are not cohesive are:

   - Comprehensibility. Classes are easier to be understand, as their is a clear determination of the functionality and structure of a class.
   - Reusability. It is expected that implementing a container class as we proposed it before will raise the possibility of and easy reuse.

8. Analysing the Site A we observed that classes that are used just to hold particular data and which have just a pair of methods to set and to retrieve that data have low TCC value, and in this case there is no possibility to split the class.

20.5 Measuring Reuse by Inheritance

Based on the findings in the previous study of metrics (see Section 20.4) and especially of coupling, we define a set of new metrics that try to eliminate the weak points of previous definitions and support re-engineering activities. This section is structured as follows: first, we discuss the motivation of our approach, and present three guidelines for the definition of metrics propoer for re-engineering. Next we introduce the concept of *multi-layered systems of metrics* and propose two such systems for the measurement of reuse by inheritance. We conclude the section by presenting the results of an experimental study based on these new metrics and taking a look at the future work that has to be done.

20.5.1 Introduction

The need of analysing and assessing object-oriented software has dramatically increased in the last years. This has lead to a large amount of research effort being invested in the area of object-oriented software measurement, both for predicting and evaluating the quality of software [BRIA 96b]. In spite of the great need for proper measurements and of the intense research efforts on object-oriented metrics, the large majority of the proposed measurements were not assimilated in the industry [BRIA 96a].

20.5.1.1 Why Do Metrics Often Not Help?

We think there are three main causes for this fact. First, there is a constant lack of following a rigourously specified process for defining software measures [BRIA 96a]. Both Fenton [FENT 97] and Whitemire [WHIT 97] emphasised this aspect proposing such rigourous processes for defining metrics, yet these processes are still not followed. The use of a standard terminology or formalism would eliminate the ambiguities found in most metrics definitions and would increase therefore their usefulness and trustworthiness [BRIA 96b] [FENT 97]. Second, metrics definitions are not only quite ambiguous, but also very dispersed. In other words, there is a lack of order among metric definitions. This order would permit to compare and group together similar metrics offering in the same time a frame in which further contributions could be easily added [BRIA 96b]. Finally, the use of software metrics is reduced because of lack of consistent experimental results for the proposed metrics. In the end, these results are the on which indicate if it is really useful to apply a particular measurement.

Therefore, our goal is to define in a rigourous manner a system of object-oriented measures, proving that it represents an adequate frame for defining further measurements that evaluate the different aspects of reuse by inheritance, and showing how this system can be applied to the re-engineering process.

20.5.1.2 Three Guidelines for Defining Proper Metrics for Re-engineering

We assert that metrics used in re-engineering must reflect deep and sophisticated characteristics of the object-oriented design, and therefore the definition of such metrics are not expected to be trivial. Three guidelines can be deduced from this principle:

- *Don't fear complex metrics!* We need to refocus our efforts from the study of trivial metrics to more complex metrics.
- *Take full advantage of the source-code!* We need to use the class interfaces, but also take advantage of their implementation; this is not available when we design the system, but it is available when we re-design it, therefore it should be used. We expect that this will lead us to definitions of metrics that are more useful for the goal we want to reach.
- *Go for the real thing!* Because most metrics are based on design-level information, they can only measure potentials of internal attributes, like coupling, cohesion or reuse. This might be enough for predicting the quality of a system based on the design, but it is insufficient for assessment of a system during a re-engineering process. Therefore, we need to use metrics that will measure the real state of the attributes, not only the potential.

We will illustrate the third guideline through an example related to the metric system that we will define next. One of the viewpoints for the NOC metric is: "The greater the number of children, the greater the reuse, since inheritance is a form of reuse."[CHID 94]. But on a careful analysis we observe that this metric is in fact not a measure of the real reuse, but of the potential reuse of the base class. In order to measure the real reuse we have to analyze the derived classes and look for the places where this reuse happens. This requires the analysis of the bodies of the child class methods; and this is an important part of measurement definitions we are going to propose next.

20.5.2 Definitions and Notation

Definition 24 (Set of Classes.) *Structurally, an object-oriented system consists of a set of classes,* C. Thus, for every class c in the system, $c \in C$.

definition At the class-level we define following concepts:

Definition 25 (Declared and Implemented Methods) *[BRIA 96b]. For each class $c \in C$ let*

- *$M(c)$ be the set of the methods from class;*
- *$M_D(c)$ be the set of* **declared methods** *in c, i.e. the virtual methods of c and the methods that c inherits but does not override;*
- *$M_I(c)$ be the set of* **implemented methods** *in c, i.e. the inherited methods that c overrides and the non-virtual, non-inherited methods of c.*

Thus, $M(c) = M_D(c) \bigcup M_I(c)$ and $M_D(c) \bigcap M_I(c) = \emptyset$.

Definition 26 (Set of Methods.) *The set of all implemented methods of a set of classes C is named set of methods, notated M(C).* Formally,

$$M(C) = \bigcup_{c \in C} M_I(c)$$

Definition 27 (Defined Class Member.) *A defined class member (dcm_c) of a class $c \in C$, is a non-inherited attribute or a implemented method of that class. The set of defined class members is denoted by $DCM(c)$ ($dcm_c \in DCM(c)$).*

Definition 28 (Accessible Class Member.) *An accessible class member (acm_c) of a class $c \in C$, is a defined class member that is accessible to other classes. The set of accessible class members is denoted by $ACM(c)$ ($acm_c \in ACM(c)$). Between the set of defined class members and the set of accessible class member exists an inclusion relation: $\forall c \in C, ACM(c) \subset DCM(c)$.*

Definition 29 (Set of Ancestors.) *The set of classes from which a class c is directly or indirectly derived is called set of ancestors of class c, notated $A(c)$.*

20.5.3 Defining a Multi-Layered System of Metrics

Definition 30 (Multi-Layered System of Metrics (MLSM)) *. A multi-layered system of metrics* (MLSM) *is a set of interdependent metric definitions organised in layers, in which the definition of each layer – excepting the lowest – is based upon the definitions of the lower layers in the system.*

The motivation behind the definition of multi-layered systems of metrics stays in the fact that most internal measurement attributes are too rich conceptually to be measured by using a single metric. This fact has caused in the past a lot of confusion.

This multi-layered approach is not new. In fact, it follows the same divide-and-conquer principle implemented as standard to the measurement of software quality [ORGA 91]. The idea of a decompositional approach stays also behind the quality models of Boehm [BOEH 78] and McCall [MCCA 77]. From this point of view, MLSM should be understood as a new way of applying the decompositional approach.

In contrast to the quality models, MLSM does *not* refer to the decomposition of higher-level quality factors in lower-level criteria [MCCA 77], but it refers to the decomposition of *metric definitions* based on the different levels where a software quality factor can be evaluated (e.g. reuse can be measured at the level of a method, but it can be also measured at the class level or system level). The measurement results at each level of measurement can and have to be interpreted differently, increasing thus the potential usefulness of the metric.

There is three-step method by which a MLSM deals with this problem:

1. **Identifying the Layers.** The different levels at which the attribute can be measured are identified and named.
2. **Defining the Layers.** The general form of each layer is defined. The identification and definition of the layers must capture the general characteristics of the attribute.
3. **Defining Layer-specific Metrics.** In order to use the system of metrics, for every layer a concrete metric definition has to be provided. The definition of these metrics is determined by the particular definition used for the attribute.

The power of this metrics system lies in its layers. These layers are a kind of slots where suitable definitions of metrics have to be ”plugged-in”. This means, that every time when we think about a new manner of computing the reuse of a class in its descendants, we are just defining a new plug-in, that will naturally belong to the system. This offers a systematic, yet flexible manner of defining and organizing metrics.

20.5.4 The ”Reuse of Ancestors” System of Metrics

The name of this MLSM speaks already about the attribute that we want to measure: the reuse of ancestors in the derived classes.

20.5.4.1 Identification and Definition of the Layers

The reuse of ancestors can be evaluated at one of the three levels:

- *Method-Ancestor level.* In fact the real reuse of ancestors takes place at this level, therefore this is the base (bottom) level of the system.
- *Class-Ancestor level.* When speaking about reuse we usually think in terms of *classes* reusing other classes.
- *Class level.* Often we need to assess the total reuse for a class (e.g. in order to evaluate its self-sufficiency). For this we need a higher-level definition of reuse.

The Bottom Layer – Reuse of Ancestor in Methods (RAM)

Definition 31 (The RAM Layer.) *Any metric that expresses the measure of reuse of an ancestor class $a \in A(c)$ at the level of the methods $m_c \in M_I(c)$ from class c, belongs to the RAM layer.* Formally, a metric belongs to this layer if it is defined as follows:

$$RAM : D \subset M(C) \times C \to \Re$$

where $D = \{(m_c, a) \mid c \in C \ \wedge \ m_c \in M_I(c) \ \wedge \ a \in A(c)\}$

The Middle Layer – Class-Ancestor Reuse (CAR)

Definition 32 (The CAR Layer.) *Any measurement that quantifies the reuse relation between a class $c \in C$ and an ancestor class, $a \in A(c)$, belongs to the CAR layer of this metrics system.* Rigourously, a metric belongs to this layer if it conforms to following definition:

$$CAR : D \subset C \times C \to \Re$$

where $D = \{(c, a) \mid c \in C \ \wedge \ a \in A(c)\}$

The Top Layer – Total Reuse of Ancestors (TRA)

Definition 33 (The TRA Layer.) *Any metric that measures the total reuse of all the ancestors for a given class $c \in C$, belongs to the TRA layer.* Formally, a metric belongs to this layer if it is defined as:

$$TRA : C \to \Re$$

Metrics defined in this layer offer a global view of the reuse in a class, allowing comparisons among different classes in the system.

20.5.4.2 Layer-Specific Metric Definitions

As we have seen in the Section 20.5.3, the layers of a MLSM define the skeleton of the system, by rigourously specifying what types of metrics can be part of the system. The next step is to define concrete metrics for each layer. We state earlier that the definition of these metrics is determined by the particular definition used for the attribute. There are mainly two kinds of reuse relation (from an ancestor) that can be measured by using this system of metrics:

- *Implementation reuse*, i.e. to take advantage in a derived class of the structure and/or behaviour implemented in an ancestor class.
- *Interface reuse*, i.e. to implement the interface defined by an ancestor class. (usually the ancestor is abstract)

Metrics for the RAM Layer. Because the bottom layer defines a direct measurement, this is the place where the distinction between the two kinds of reuse will be made. Although, in this paper we decided to define only metrics for the *implementation reuse*, metrics that measure the interface reuse can be defined in the same manner.

Metric 1 Unitary RAM Metric *The metric defines the reuse of an ancestor class $a \in A(c)$ in a method mt_c of class $c \in C$ as follows:*

$$RAM_{unit}(mt_c, a) = \begin{cases} 1 & \text{if } \{acm_a \mid acm_a \in ACM(a) \wedge mt_c \text{ uses } acm_a\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The RAM_{unit} metric identifies the methods that use a member of an ancestor, but it does neither express the intensity of the reuse, nor does it differentiate between using the interface or the implementation of the ancestor class. These two aspects are covered in the next definition.

Metric 2 Percentage RAM Metric. *The metric defines the reuse of an ancestor class $a \in A(c)$ in a method mt_c of class $c \in C$ as follows:*

$$RAM_{perc}(mt_C, a) = \frac{\sum_{i=1}^{n_a^{imp}} uses(acm_i^{imp}, mt_c) + (1 - k_a) \cdot \sum_{j=1}^{n_a^{int}} uses(acm_j^{int}, mt_c)}{n_a^{imp} + (1 - k_a) \cdot n_a^{int}}$$

where acm_i^{imp} ($i = \overline{1, n_a^{imp}}$) are the accessible class members of the ancestor class a that belong to the implementation of the class, acm_j^{int} ($j = \overline{1, n_a^{int}}$) are the accessible class members of ancestor a that belong to the interface of the class, and $uses$ is defined as:

$$uses(acm_a, mt_c) = \begin{cases} 1 & \text{if } mt_c \text{ uses } acm_a \\ 0 & \text{otherwise} \end{cases}$$

In spite of the complicated formal aspect the metric is quite simple. The metric is mainly composed of two sums: the first sum is counting how many *distinct* members, belonging to the implementation of an ancestor class $a \in A(c)$, are used in a method of class c, while the second sum counts the members from the interface of the ancestor that are used in that method. Because reuse of implementation is in fact a form of coupling and because the interface of a class is considered being much stabler than the implementation, a *stability factor* $k_a \in [0, 1]$ has been introduced in order to consider this difference when computing the metric. The bottom part of the fraction is included only for the normalisation of the metric.

Metrics for the CAR Layer

Metric 3 CAR Metric. *The CAR metric defined between a class $c \in C$ and one of its ancestor classes $a \in A(c)A$ is formally expressed as:*

$$CAR(c, a) = \frac{\sum_{i=1}^{n_c} RAM(mt_i, a)}{n_c}$$

where $mt_i \in M_I(c)$, $i = \overline{1, n_c}$ and $n_c = cardM_I(c)$.

The CAR metric defined above quantifies the reuse from an ancestor by summing the reuse in all the implemented methods of the class. The way reuse is calculated at the method-level is encapsulated in the bottom layer, RAM. Thus, the CAR metric can be computed using any of the two previously defined RAM metrics, or any other metric that belongs to the RAM layer.

Notice that the values of this metric are again *relative* numbers, like in the metrics defined for implementation reuse in the RAM layer. We define the CAR metric this way because we considered them more adequate for the measurement of implementation reuse. If we were to define the metric thinking on the interface reuse, we would have probably defined it using an absolute range of values. We should also remark that this metric respects the definition given for a MLSM, as its definition is based on the lower layer of the system.

Metric for the TRA Layer

Metric 4 TRA Metric. *The TRA metric for a class $c \in C$ is defined as the sum of the reuses of all the ancestors of that class.* This can be formally expressed as:

$$TRA(c) = \sum_{i=1}^{n_a} CAR(c, a_i)$$

where $a_i \in A(c)$,$i = \overline{1, n_c}$ and $n_c = cardM_I(c)$.

TRA defined in this manner offers a simple way to compute a total reuse value for a class, based on the metric defined for the middle layer CAR. This is for sure not the only possibility to define TRA. For a particular measurement goal, it would be possible to calculate TRA based directly on the definition ofRAM, or to differentiate in the sum between ancestors.

20.5.5 The ”Reuse in Descendants” System of Metrics

As the name suggests, this measurement is about the reuse of a base classes in the classes derived from it. The reuse of a class by another class is a form of coupling. In defining this metric we evaluate coupling from the perspective of the server class, we say that it is a measurement of export coupling [BRIA 96a]. In [MARI 99] we have defined a complementary metrics-system called ”Reuse from Ancestors”. That metrics system measures the inheritance-based reuse from the perspective of the descendant class.

20.5.5.1 Identification of the Layers

We identified three levels (layers) on which the reuse of a class in its descendants can be evaluated:

- *Base Member - Descendant Class.* In fact a descendant class is using some of the members (methods and sometimes attributes) defined in the base class. The real reuse occurs at this level, therefore this is the bottom layer, called *RMD layer* (**R**euse of **M**ember in **D**escendant-class).
- *Base Class - Descendant Class.* When speaking about reuse, we usually think in terms of a class being used by another class (server perspective); therefore we need to define this middle layer that we called *RID layer* (**R**euse **I**n **D**escendant-class)
- *Base Class.* Often we want to assess the total reuse of a class in all its descendants. We named this the *TRID layer* (**T**otal **R**euse **I**n **D**escendant-class).

20.5.5.2 Layer-Specific Metric Definitions

Next we define for each layer the metrics ("plug-ins") that we used in our experimental study, pointing out to alternative manners of defining these metrics.

Metrics for the Bottom Layer – RMD

Definition 34 (Reuse of Members in Descendant-class – RMD) . *We define RMD as the relative number of methods from the descendant-class D in which a member m_C of class C is used:*

$$RMD(m_C, D) = \frac{1}{n_D} \cdot \sum_{i=1}^{n_D} uses(m_C, mth_D^{(i)})$$

where n_D is the number of methods in class D, and the $uses$ function is defined as:

$$uses(m_C, mth_D) = \begin{cases} 1 & \text{if class-member } m_C \text{ is used in method } mth_D \\ 0 & \text{if class-member } m_C \text{ id not used in } mth_D \end{cases}$$

As mentioned earlier, this is not the only possible definition for RMD. For example we could have defined the metric in a binary manner: 1 if the member is used in any methods of the derived class, and 0 if it is not used at all.

Metrics for the Middle Layer – RID

Definition 35 (Reuse In Descendant-class – RID) . *The RID metric expresses the reuse of a base-class C in one of its descendant classes D, as the average use of all the usable members of C. This can be formally expressed as:*

$$RID(C, D) = \frac{1}{n_C} \cdot \sum RMD(m_C^{(i)}, D)$$

where n_C represent the number usable members defined in class C.

Observation: The way the reuse of a member is calculated is not directly specified in the definition of RID; indeed the metric bases its definition on the lower layer RMD where the reuse of member in a descendant class is defined.

Metrics for the Top Layer – TRID

Definition 36 (Total Reuse In Descendants – TRID) . *We define the $TRID$ metric for a class C as the sum of the RID values calculated between class C and all its descendants. This can be formally expressed as:*

$$TRID(C) = \sum_{i=1}^{n_D} RID(C, D^{(i)})$$

where n_D represents the number of descendant classes for class C.

The above definition is calculated based on the reuse in each descendant (RID). A member-based definition of TRID might be an alternative to this definition. In that case we would add together the reuse of each member in all the descendants, based on RMD.

20.5.6 The Case Studies

Before presenting the results of the experimental study we will shortly describe the three case-studies, both in a descriptive and a quantitative form (see Table 20.11) in order to offer a better understanding of the study and permit some possible correlations between the metrics results and the projects.

Site A is a part of a project that aims to produce software that supports the development of control and command systems. The software was developed for *Windows NT* using MFC/VisualC++. In the frame of this project a number of frameworks were used, e.g. ObjectStore, MFC, and some own frameworks of the company that developed the software system. *Site A* is in fact the part of the project where the *reusable classes* were stored. (Reusable Class Pool). The project was developed over a period of 4 years and involved approximately 15-20 software developers.

Site B is an application for graphical visualisation of measurement values. It is based on the Microsoft MDI framework and supports graphical views in the space and time domain. For time variant measure values animated diagrams are supported. It implements several graphical objects to display the measure values and additional graphical information objects. The run-time optimised display of animated diagrams and the flexibility to expand the programs display capacity by adding new graphical objects was one of the major design goals. The development of the project lasted one and a half years and it was designed and implemented by 4-5 people, for the Windows NT platform. *Site B* is still under permanent extension and enhancement.

Site C is part of a very large project in which a number of 3-4 important companies and research institutes from USA and Germany were involved. The application is the development of a tracking-software that simulates and computes the trajectories of sub-atomic particles, obtained through a particle accelerator. The software was developed under different UNIX systems (IRIX, Linux, OSF/1) using the KAI C++ compiler[3] The project begun in 1997 and a second version is now almost finished. The number of people working on this software is around 12, and the programmers do not have much experience on programming in C++.

Site Name	Source code [in KB]	Number of classes	Number of inheritance trees	Maximum depth of inheritance tree
Site A	1258	228	45	9
Site B	500	69	12	3
Site C	4620	323	49	5

Table 20.11: Overview of the case studies

[3]The KAI C++ compiler is strictly respecting the standard C++ language definition.

Observation: For this experimental study we could have a direct contact only to the developers of *Site B*. For the other two sites we tried to use directly the information from the source code, but this is in most of the cases not enough. We plan in the next tine to contact the developers of *Site C* and *Site A* hoping that we will get in this way more precious information for the experimental validation of these metrics.

20.5.7 Experimental Evaluation of the "Reuse of Ancestors" Metric System

Based on the definitions presented in the previous chapter, below are the results of the two proposed definitions for the RA metric together with some experimental observations and their interpretation.

20.5.7.1 Viewpoints

1. A low average TRA value for a project might be a sign of poor object-oriented design, that does not take advantage of inheritance-based reuse.
2. If two or more classes have high CAR values in relation to a common ancestor then these classes might constitute a *subsystem* in the project.
3. The outliers with high TRA values lie deeper in the class hierarchy and are mostly *leaf-classes*. We expect the TRA values to increase from the root to the leaves of the class hierarchy.
4. The highest CAR values appear on *direct* inheritance relations.

20.5.7.2 Results Summary

Measurements Based on the RAM-Unitary Metric. The summary of the results obtained on the three case-studies when using the RAM_{unit} metric in the bottom-layer of the metrics system is presented below. The table contains the minimum the average and the maximum values for each of the three sites.

Site Name	Minimum	Average	Maximum
Site A	0.02	0.27	1.0
Site B	0.07	0.47	1.0
Site C	0.02	0.35	0.81

Table 20.12: Results summary for the *CAR* metric based on *RAM-Unitary*

Site Name	Minimum	Average	Maximum
Site A	0	0.38	1.90
Site B	0.10	0.56	1.00
Site C	0.10	0.61	1.10

Table 20.13: Results summary for the *TRA* metric based on *RAM-Unitary*

Measurements Based on the RAM-Percentage Metric. The summary of the results obtained on the three case-studies when using the RAM_{perc} metric in the bottom-layer of the metrics system is presented below. The table contains only the average and the maximum values for each of the three sites, because the minimum values can be approximated with zero.

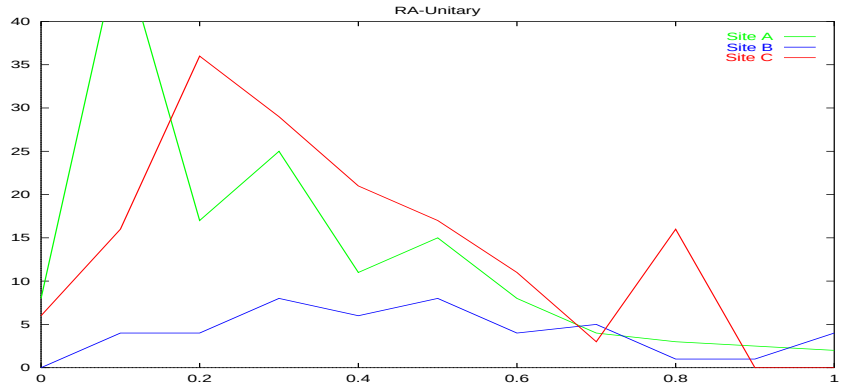


Figure 20.11: Comparative Graphic for RA-Unitary

	Site Name	$k_a = 0.0$	$k_a = 0.5$	$k_a = 1.0$
Average	Site A	0.02	0.03	0.06
	Site B	0.11	0.13	0.24
	Site C	0.02	0.02	0.01
Maximum	Site A	0.17	0.23	0.40
	Site B	0.33	0.42	0.58
	Site C	0.13	0.16	0.18

Table 20.14: Results summary for the *CAR* metric based on *RAM-Percentage*

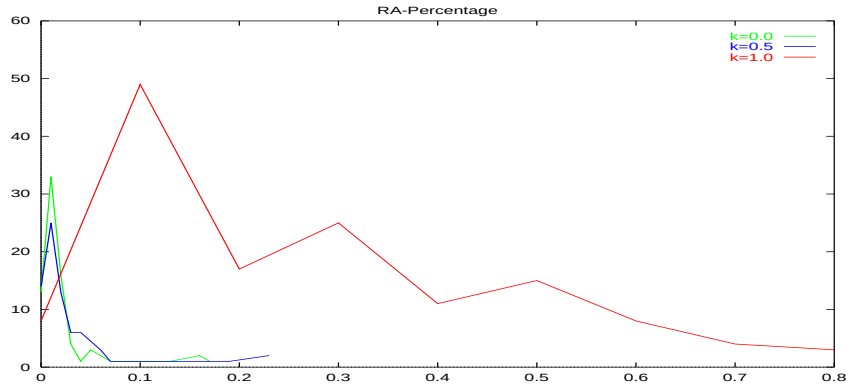


Figure 20.12: CAR Graphic for Site A, based on *RAM-Percentage* using different values for the stability factor k

20.5.7.3 Results Analysis

In all the three case studies the Top 10 outliers both for $CAR - Percentage$ and $TRA - Percentage$ do not change very much when varying the stability factor k_a between $k_a = 0$ and $k_a = 0.5$; but for *Site B* and *Site C* the top did change very much when the ancestor class was considered completely stable ($k_a = 1.0$). Comparing the values for $CAR - Percentage$ for different values of the stability factor for all the three case-studies the order was: $CAR_{perc}^{k_a=0} < CAR_{perc}^{k_a=0.5} < CAR_{perc}^{k_a=1.0}$. Thus the highest factor of code reuse (don't forget these are relative values) was obtained considering the superclass stable and ignoring its interface. The conclusion is that there are a lot of public members in the superclass, but few of them are reused through inheritance relations.

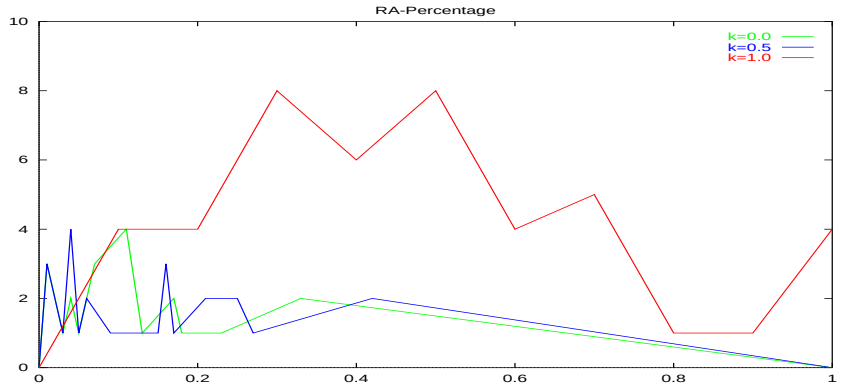


Figure 20.13: CAR Graphic for Site B, based on *RAM-Percentage* using different values for the stability factor k

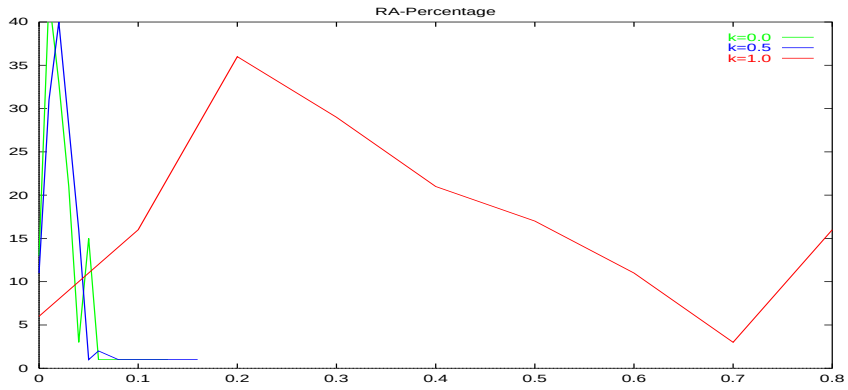


Figure 20.14: CAR Graphic for Site C, based on *RAM-Percentage* using different values for the stability factor k

	Site Name	$k_a = 0.0$	$k_a = 0.5$	$k_a = 1.0$
Average	Site A	0.04	0.05	0.10
	Site B	0.13	0.16	0.20
	Site C	0.04	0.04	0.03
Maximum	Site A	0.24	0.29	0.40
	Site B	0.33	0.42	0.58
	Site C	0.13	0.16	0.18

Table 20.15: Results summary for the *TRA* metric based on *RAM-Percentage*

Almost all outliers for total $TRA - Percentage$ are *"light-classes"*, i.e. classes with a small number of methods. This could also be an interesting point for *model capture*.

The Top 10 for the $CAR - Unitary$ strongly differs from the $CAR - Percentage$ top (20% for *Site A*, 30% for *Site B*, 20% for *Site C*). This demonstrates that the two definitions are not correlated, indicating that classes where an ancestor is reused in many methods($RAM - Unitary$), are not the same with the classes where the reuse is more intense($RAM - Percentage$). Thus, we deduce that classes that reuse much of an ancestor class, do it in a small number of methods.

Concerning the average values for the three analysed projects, we found a correlation between these values and the maximum depth of inheritance tree: the deeper the class hierarchy, the smaller the code reuse from

ancestors.

20.5.7.4 Viewpoints Validation

1. *First*. Analysing the results presented in Table 20.12 and Table 20.14 and based on supplementary information concerning the quality of the design, we can confirm that *Site A* is not as well designed compared with the other two sites, sustaining our first viewpoint.

2. *Second*. Concerning our second viewpoint, on *Site A* and *Site B*, when using the $RAM - Unitary$, the first half of the top consists of classes that can be *clustered*. Discussions with the developers of the systems confirmed that those clusters are indeed *subsystems*. It still has to be studied if this can be used as a general rule for model capture. Another question that still remains open is, what kind of subsystems are those detected with these metrics

3. *Third*. For all case studies the very most CAR relations involve a leaf-class (*Site B* – 76.0%; *Site B* – 95.8%; *Site C* – 79.3%), confirming the third viewpoint.

4. *Fourth*. The hypothesis that the highest CAR values are between classes that are in a *direct* inheritance relation proved to be true for *Site B* and *Site C* (83.3% and 73%). In *Site A* we encountered an interesting a class, having 55 descendants spread on 4 hierarchy levels. The class is reused by most of its indirect descendant classes, so that approximately the half (47.1%) of the inheritance-reuse relations are indirect.

20.5.8 Experimental Evaluation of the ”Reuse in Descendants” Metric System

20.5.8.1 Viewpoints

1. A base-class is used either to export code to its descendants or to define an interface. For the classes that export code we expected high TRID values, and for those defining interfaces without offering much implementation we expect lower values.

2. From the viewpoint of maintainability, the classes that cause most maintenance efforts are the most used ”server” classes. In respect to inheritance, these classes should be the outliers of the $TRID$ metric. Because these classes are intensively reused in the subclasses, they should be kept stable, avoiding chains of changes through the class hierarchy.

3. The outliers will indicate central classes, because we suppose that classes offering a lot of reusable services must play a central role in the design of the application.

4. We expect class-pairs with high RID values to result from direct inheritance relations. If this is not the case the ancestor class is probably an important and powerful abstraction, because it’s code is reused in classes that are positioned deeper in the class hierarchy.

20.5.8.2 Results Summary

We summarized the results of the RD metric for the case were we use the RDD-Unitary definition to parameterize the metric. The table contains the minimum the average and the maximum values for each of the three sites.

Site Name	Minimum	Average	Maximum
Site A	0.00	0.02	0.17
Site B	0.01	0.11	0.31
Site C	0.00	0.02	0.13

Table 20.16: Results summary for the Class-Descendant class (RD) layer

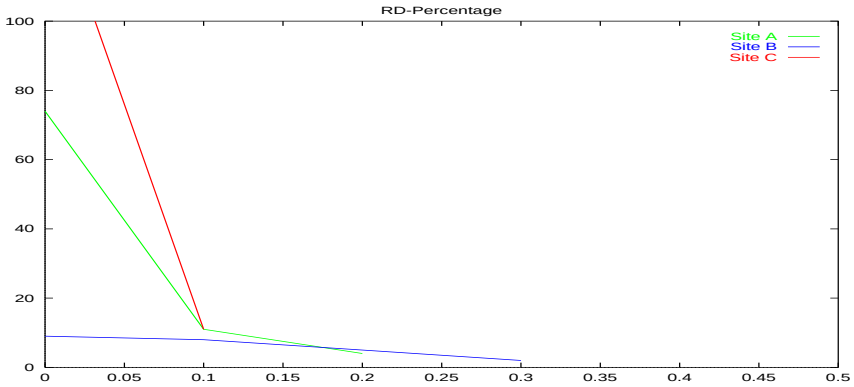


Figure 20.15: Comparative Graphic for RMD metric

Site Name	Minimum	Average	Maximum
Site A	0	0.48	1.7
Site B	0	0.64	1.3
Site C	0	1.19	22.5

Table 20.17: Results summary for the Total Reuse In Descendant ($TRID$) layer

Site Name	Perc. of Base-classes reused by descend.	Number of inheritance trees	Average depth of inheritance tree
Site A	38%	45	5.4
Site B	57%	12	2.5
Site C	40%	49	4.2

Table 20.18: Correlation between the percentage of reused base-classes and the structure of the hierarchy forest

20.5.8.3 Results Analysis

A correlation was observed between the percent of super-classes that are reused in code on the one hand and the number of inheritance trees, and the average depth of the class hierarchy on the other hand (although the last two are not absolutely independent measures). This correlation (Table 20.18) shows that a large class-hierarchy structure reduces the code reuse from base-classes.

Also in Site B a class was detected that had a significantly lower TRID value than expected by the designer. The cause was a number of attributes of the class that were declared protected although conceptually they were private. If the attributes would have been declared private the class would have moved among the first three outliers for this metric!

Observation: We think that the second experimental observation speaks in favor of the metric, because it proves that the metric can be used by different people (e.g. the designer of the system and a person re-engineering a foreign system) in different manners. The above observation could not have been made by anyone else but the designer himself. This brings us to the more general idea that viewpoints for a metric must be defined keeping in mind the different categories of people that might use the metric. Of course, some of these viewpoints will be common, but some might be specific. On the other hand, that experimental observation addressees one time more the issue of the reliability of measurement results, i.e the way these results might be altered by small mistakes like the one mentioned before.

20.5.8.4 Viewpoints Validation

1. *First.* In all the three case-studies, almost all base classes in the top-ten outliers are classes defining interfaces (although they were not abstract classes), proving that our initial assumption was wrong. The main cause is the fact that we didn't consider polymorphic potential of the reused base classes. If a base class includes virtual methods - and a lot of them do so - the reuse of those methods in the derived classes can't be counted as a real code reuse of the base-class, although statically the invoked methods appear as belonging to the base class of the hierarchy. Still the observation is an interesting point for re-engineering, especially for model capture.

2. *Second*. In Site B the designers agreed on the fact that the TRID outliers were the classes with a major impact on maintenance, but they also remarked that no maintenance problems appeared because of these classes as they were from the very beginning properly designed. For the other two case-studies we have not yet enough information on the maintenance effort.

3. *Third*. Concerning this hypothesis we observed that the TRID outliers are not central classes from the functionality point of view. They are rather small subclasses that "sketch" the different abstractions that are concretely implemented in the subclasses. This observation strengthens our belief that what we detected were not the base-classes that are mostly reused, but the base-classes with the highest polymorphic potential.

4. *Fourth*. For Site A and Site B all the outliers come from direct inheritance relations. In Site C a class was detected that was reused by a large number of subclasses that were not direct descendants of that class. For this case the class proved to be indeed a powerful abstraction, but there is still a need for more evidence in order to confirm the generality of the hypothesis.

20.5.8.5 Conclusions on "Reuse in Descendants"

Not every object-oriented metric can be usefully applied to support re-engineering operations. In order to use metrics for that, the metrics must conform to a number of criteria. The system of metrics that we proposed in this paper offers a flexible and systematic manner of defining and organizing metrics that measure the reuse of a base class in its descendant classes. The concrete metrics that we propose are intended to measure the real reuse and not only a potential one.

Although the experimental study is not yet complete, there is one important lesson that we learned so far: in measuring the real reuse of a base class we have to explicitly define how to deal with polymorphic methods. Some of the experimental observations we made so far in respect to model capture are encouraging , but still a lot of work has to be done in order to achieve solid conclusions.

The future work includes first of all a sharpening of the definition used at the bottom layer (RMD) so that it explicitly deals polymorphic methods. We also intend to do a comparative experimental study between this metrics system other related inheritance metrics, in order to find out a possible correlation between them. A third research direction is to study the possibility of using these metrics for problem detection.

20.5.9 Conclusions

This work is only a first step in fulfilling a more comprehensive vision on using metrics as a support for re-engineering. Therefore no final conclusions can be stated at this point. But this study has helped us to learn some important things that will be mentioned next.

The most important conclusion of this study is that metrics, if defined properly, *can* be successfully used to support re-engineering activities. In this study we have a clear evidence of the fact that the main cause of the partial failure in using metrics for this kind of activities in the past, was the use of unproperly defined metrics. When moving from trivial size-metrics to more complex metrics the chances of detecting design flaws or even capturing the major elements of the initial design strongly increase.

Secondly, we understood the importance of bringing order among the metrics, by organising them in *multi-layered systems of metrics* as we did it with the inheritance based metrics. At this point we are strongly convinced that the multitude of metrics defined until now *can* and *should* be organised in this manner. This approach of metrics would oblige us to find for each metric that place it takes in the ”metrics orchestra”, giving us a clear understanding of its role and potential utility. Our next goal is to organise the metrics that measure the behavioural coupling in such a multi-layered system.

A third conclusion is this metrics suite proved to be very efficient in finding the major elements of the design. In other words these metrics proved to be very efficient in *model capture.* Concerning their utility for *design flaw detection*, although there are some signs that indicate this, still more practical experience is requested for a correct conclusion.

A last conclusion, – but not less important – refers to the importance of a rigourous validation of metrics. The continuation of this work must necessarily include a theoretical validation of the proposed system of metrics together with a continuation of the experimental studies (empiricial validation). The objective of the theoretical validation will be to prove if the proposed metrics do accurately measure the attributes they were designed to measure. For this, we intend to focus our attention on the criteria proposed by Kitchenham et al. [KITC 95] and on the properties defined by Briand et al. [BRIA 96a]. Concerning the empirical validation we intend to conduct a rigourous experimental study, based several on real-world legacy systems, aiming to investigate the usefulness of these metrics especially for design-flaws detection in legacy systems. We believe that these experiments must rely on the principles of empirical investigation [FENT 97].

Appendix A

Glossary

Class versioning.
Each change to a class results in a new, distinct class definition; each object belongs to one class version, which is thus a snapshot of the class definition at a certain point in the life of a system [BJOR 89].

Conversion.
The physical storage format of objects is transformed to match a different class definition [LERN 90].

Design Pattern.
A proven design that describes the core of a solution to a problem which occurs over and over again in OO software design, together with its range of applicability. The solution usually has developed and evolved over time [GAMM 95].

Design recovery.
A subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the the subject system. The objective of design recovery is to identify meaningful higher-level abstractions beyond those obtained directly by examining the system itself [CHIK 90].

Filtering and screening.
Objects are wrapped with exception handlers that hide differences between different versions of the same class [SKAR 86].

Forward engineering.
The traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system [CHIK 90].

Framework.
A framework is an abstract object-oriented design together with a library of standard software components (abstract classes and templates as well as concrete components) that can be specialised, instantiated and combined to construct a number of systems with similar properties for a specific application domain [CASA 96] [JOHN 88].

Global reorganisation of hierarchies.
Algorithms put inheritance hierarchies in a normal form that minimises the number of classes and relationships and suppresses redundant definitions [CHEN 96] [LIEB 91].

Hierarchy maintenance.
Guidelines and semi-automatic algorithms are provided to reorganise inheritance hierarchies for improved reusability [PUTK 94].

Incremental reorganisation of hierarchies.
Algorithms analyse and reorganise subclassing relationships to eliminate the need for redefinitions in inheritance relationships and to factor common functionality in a hierarchy [CASA 92] [DICK 96].

Law of Demeter.
Reference and calling dependencies between methods and variables are changed to follow a modular programming style [LIEB 88].

Legacy System.
A system is called legacy if it exhibits the following properties: It is a production system carrying out business-critical tasks. It has been developed with older technology, or with older versions of an existing technology. It can no longer be easily adapted to meet changing requirements.

Method factorisation.
Code fragments common to several methods are extracted and put in separate methods to maximise code sharing [OPDY 90].

Metrics-based analysis.
A set of quality statistics on oo code is computed and then used to detect design problems and guide re-engineering activities.

Object-oriented views.
An abstraction layer is inserted between subsystems in an oo design to limit the scope of changes to single subsystems and to map different versions of subsystems to each other [BRAT 92] [RA 95].

Pattern restructuring.
Modification primitives on oo programs structured according to specific kinds of patterns and architectures are formally specified to ensure the preservation of behaviour [HÜ 96].

Pattern-directed re-engineering.
Standard software structures serve to document and analyse oo applications, and serve as targets structures for re-engineering [YELL 96] [ZIMM 95].

Redocumentation.
A form of restructuring where the resulting semantically-equivalent representation is an alternative view intended for a human audience [CHIK 90].

Reengineering.
The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form [CHIK 90].

Refactoring.
Frequent high-level re-engineering operations are identified and specified to preserve class behaviour across modifications [JOHN 93] [OPDY 90].

Restructuring.
A transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and the external behaviour of the original [CHIK 90].

Reverse engineering.
The process of analysing a subject system with two goals in mind:

1. to identify the system's components and their interrelationships; and,
2. to create representations of the system in another form or at a higher level of abstraction [CHIK 90].

Schema evolution.
A set of operations specifies the kinds of high-level revisions to an oo design that occur during conceptual modelling [LI 96].

Schema modification primitives.
A set of elementary modification operations, all specified so as to preserve basic integrity constraints, suffices to define any other more complex modification operation [BANE 87b] [ZICA 92].

Software maintenance.
Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment [IEEE 83].

Tailoring and excuses.
Language mechanisms allow to redefine inherited properties and to accommodate exceptions in specialisation hierarchies [BORG 88].

Transposed files.
By storing object variables in different tables, adding, updating and deleting variables is possible without reformatting records [ANDA 91].

Visual analysis of oo software.
Static and dynamic properties of oo programs are represented graphically to identify design and implementation problems [PAUW 93] [SEFI 96].

Bibliography

[CDI 94] *CDIF Framework for Modelling and Extensibility*, January 1994. (p 250)

[IEE 92] *IEEE Standard for a Software Quality Metrics Methodology*, 1992. (p 250)

[ISO 98] *Software Product Evaluation*, 1998. (pp 250, 253)

[ABOW 97] G. Abowd, L. Bass, P. Clements, R. Kazman, L. Northrup, and A. Zaremski. *Recommended Best Industrial Practices for Software Architecture Evaluation*. Technical Report CMU/SEI 96-TR-025, Carnegie Mellon University, 1997. (p 250)

[ABRE 96] F. B. Abreu and W. Melo. *Evaluating the Impact of Object-Oriented Design on Software Quality*. Proceedings of Symposium on Software Metrics METRICS '96, March 1996. (p 250)

[ADAM 90] R. Adamov and L. Richter. *A Proposal for Measuring the Structural Complexity of Programs*. Journal of Systems and Software, pages 55–70, September 1990. (p 261)

[AGES 94] O. Agesen. *Constrained-Based Type Inference and Parametric Polymorphism*. In Proceedings of the First International Static Analysis Symposium (SAS '94), volume 864 of *LNCS*. Springer-Verlag, 1994. (p 212)

[AGES 95] O. Agesen. *The Cartesian Product Algorithm*. In W. Olthoff, editor, Proceedings ECOOP'95, LNCS 952, pages 2–26, Aarhus, Denmark, August 1995. Springer-Verlag. (pp 212, 213)

[ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. Design Patterns in Smalltalk. Addison-Wesley, 1998. (pp 175, 181)

[ANDA 91] J. Andany, M. Léonard, and P. C. *Management of Schema Evolution in Databases*. In 11th VLDB Proceedings, pages 3–20, September 1991. (p 317)

[ARNO 92] R. S. Arnold. Software Reengineering. IEEE Computer Society Press, Los Alamitos, CA, 1992. (pp 8, 254)

[ATKI 98] G. Atkinson, J. Hagemeister, P. Oman, and A. Baburaj. *Directing Software Development Projects with Product Metrics*. In Proceedings of the 5th International Software Metrics Symposium (METRICS'98), pages 193–204, Los Alamitos, 1998. IEEE CS Press. (p 250)

[AVRI 99] A. Avritzer. *Environment for Software Assessment*. In Workshop on Object-Oriented Architectural Evolution, 13th European Conference on Object-Oriented Programming (ECOOP '99), Lisabon, Portugal, June 1999. (p 250)

[BAIL 88] W. G. Bail and M. V. Zelkowitz. *Program Complexity Using Hierarchical Abstract Computers*. Computer Language, vol. 13, no. 3/4, pages 109–123, March/April 1988. (p 261)

[BAKE 92] B. S. Baker. *A Program for Identifying Duplicated Code*. Computing Science and Statistics, vol. 24, pages 49–57, 1992. (pp 186, 187)

[BAKE 95] B. S. Baker. *On Finding Duplication and Near-Duplication in Large Software Systems*. In Proc. Second IEEE Working Conference on Reverse Engineering, pages 86–95, July 1995. (pp 187, 237)

[BALL 96] T. Ball and S. Erick. *Software Visualization in the Large*. IEEE Computer, pages 33–43, 1996. (pp 31, 33, 137, 241)

[BANE 87a] J. Banerjee, H.-T. Chou, J. F. Garza, W. Kim, D. Woelk, N. Ballou, and H. Kim. *Data Model Issues for Object-Oriented Applications*. ACM TOOIS, vol. 5, no. 1, January 1987. (p 100)

[BANE 87b] J. Banerjee, W. Kim, H.-J. Kim, and H. Korth. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*. In Proceedings ACM SIGMOD '87, pages 311–322, December 1987. (p 317)

[BÄR 98] H. Bär and O. Ciupke. *Exploiting Design Heuristics for Automatic Problem Detection*. In S. Ducasse and J. Weisbrod, editors, Proceedings of the ECOOP Workshop on Experiences in Object-Oriented Re-Engineering, number 6/7/98 in FZI Report, June 1998. (pp 201, 213)

[BASI 95] V. Basili and W. M. L. Briand. *A Validation of Object-Oriented Design Metrics as Quality Indicators*. Technical Report, University of Maryland, April 1995. (pp 273, 284, 285)

[BAUE 98] M. Bauer. Reengineering von Smalltalk nach Java. Master's thesis, Institut für Algorithmen und Datenstrukturen, Universtät Karlsruhe, 1998. (pp 213, 214)

[BAUE 99] M. Bauer. *Analyzing Software Systems by Using Combinations of Metrics*. In O. Ciupke and S. Ducasse, editors, Proceedings of the ECOOP'99 Workshop on Experiences in Object-Oriented Re-Engineering, number 26/6/99 in FZI Report, June 1999. (pp 21, 27)

[BAXT 98] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. *Clone Detection Using Abstract Syntax Trees*. In Proceedings ICSM 1998, 1998. (p 237)

[BECK 94] K. Beck. *Death to Case Statements*. Smalltalk Report, pages 8–9, January 1994. (pp 175, 181)

[BECK 97] K. Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1997. (pp 9, 66, 69, 71, 111, 141, 178)

[BIEM 95] J. M. Bieman and B. K. Kang. *Cohesion and Reuse in an Object-Oriented System*. Proceedings of the ACM Symposium on Software Reusability, April 1995. (pp 25, 206, 279, 280)

[BIRD 88] R. Bird and P. Wadler. Introduction to Functional Programming. International Series in Computer Science. Prentice Hall, 1988. (p 87)

[BJOR 89] A. Bjornerstedt and C. Hulten. *Version Control in an Object-oriented Architecture*. In W. Kim and F. Lochovsky, editors, Object-Oriented Concepts, Databases and Applications, pages 451–485. Addison-Wesley/ACM Press, Reading, Mass., 1989. (p 315)

[BOEH 78] B. Boehm and J. Kapsar. *Characteristics of Software Quality*. TRW Series of Software Technology, Amsterdam, North Holland, 1978. (p 301)

[BOOC 94] G. Booch. Object Oriented Analysis and Design with Applications. The Benjamin Cummings Publishing Co. Inc., 2nd edition, 1994. (p 105)

[BOOC 98] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. Addison-Wesley, 1998. (p 9)

[BORG 88] A. Borgida. *Modelling Class Hierarchies with Contradictions*. In SIGMOD Record (special issue on SIGMOD '88), pages 434–443, September 1988. (p 317)

[BRAT 92] S. E. Bratsberg. *Unified Class Evolution by Object-Oriented Views.* PhD thesis, October 1992. (p 316)

[BRIA 96a] L. Briand, S.Morasca, and V. Basili. *Property-Based Software Engineering Measurement*. IEEE Transactions on Software Engineering, vol. 22, no. 1, January 1996. (pp 266, 268, 299, 300, 304, 313)

[BRIA 96b] L. Briand, J. Daly, and J.Wuest. *A Unified Framework for Coupling Measurement in Object-Oriented Systems*, 1996. (pp 299, 300, 301)

[BRIT 95] F. Brito e Abreu. *The MOOD Metrics Set*. Proc. ECOOP'95 Workshop on Metrics, 1995. (p 267)

[BROD 95] M. Brodie and M. Stonebraker. Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach. Morgan Kaufman, 1995. (p 164)

[BROO 87] F. P. Brooks. *No Silver Bullet*. IEEE Computer, vol. 20, no. 4, pages 10–19, April 1987. (pp 121, 130)

[BROW 96] K. Brown. *Design Reverse-engineering and Automated Design Pattern Detection in Smalltalk*. Research Report TR-96-07, North Carolina State University, 1996. (pp 9, 124)

[BROW 98] W. J. Brown, R. C. Malveau, H. W. S. McCormickIII, and T. J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. Wiley and Sons, 1998. (pp 11, 134, 164, 185, 186)

[BUSC 96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stad. Pattern-Oriented Software Architecture – A System of Patterns. John Wiley, 1996. (pp 92, 124)

[CASA 91] E. Casais. *Managing Evolution in Object Oriented Environments: An Algorithmic Approach.* Ph.D. thesis, Centre Universitaire d'Informatique, University of Geneva, May 1991. (p 10)

[CASA 92] E. Casais. *An Incremental Class Reorganization Approach.* In O. L. Madsen, editor, Proceedings ECOOP'92, LNCS 615, pages 114–132, Utrecht, The Netherlands, June 1992. Springer-Verlag. (pp 10, 100, 316)

[CASA 93] E. Casais. Object-Oriented Systems, volume 1, chapter Automatic Reorganization of Object-Oriented Hierachies: A Case Study, pages 95–115. 1993. (p 100)

[CASA 94] E. Casais. *Automatic Reorganization of Object-Oriented Hierarchies: A Case Study.* Object-Oriented Systems, vol. 1, no. 2, pages 95–115, December 1994. (p 10)

[CASA 95a] E. Casais. *Managing Class Evolution in Object-Oriented Systems.* In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pages 201–244. Prentice Hall, 1995. (p 10)

[CASA 95b] E. Casais. Object-Oriented Software Composition, chapter Managing Class Evolution in Object-Oriented Systems, pages 201–244. Prentice Hall, 1995. (p 100)

[CASA 96] E. Casais. An Experiment in Framework Development. John Wiley & Sons, October 1996. (p 315)

[CASA 97] E. Casais and A. Taivalsaari. *Object-Oriented Software Evolution and Re-engineering (Special Issue)*. Journal of Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 233–301, 1997. (p 8)

[CASA 98] E. Casais. *Re-Engineering Object-Oriented Legacy Systems.* Journal of Object-Oriented Programming, vol. 10, no. 8, pages 45–52, January 1998. (p 11)

[CHEN 96] J.-B. Chen. *Generation and Reorganization of Subtype Hierarchies*. In JOOP, vol. 8, no. 8, pages 26–35, January 1996. (p 315)

[CHID 91] S. R. Chidamber and C. F. Kemerer. *Towards a Metrics Suite for Object Oriented Design*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, pages 197–211, November 1991. (p 274)

[CHID 94] S. R. Chidamber and C. F. Kemerer. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, vol. 20, no. 6, pages 476–493, June 1994. (pp 23, 24, 25, 26, 206, 209, 271, 280, 282, 300)

[CHIK 90] E. J. Chikofsky and J. H. Cross II. *Reverse Engineering and Design Recovery: A Taxonomy.* IEEE Software, pages 13–17, January 1990. (pp 10, 15, 107, 315, 316)

[CHIK 92] E. J. Chikofsky and J. H. Cross. *Reverse Engineering and Design Recovery: A Taxonomy*. In R. S. Arnold, editor, Software Reengineering, pages 54–58. IEEE Computer Society Press, 1992. (p 10)

[CHUR 95] N. I. Churcher and M. J. Shepperd. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, vol. 21, no. 3, pages 263–265, March 1995. (pp 23, 270)

[CIUP 97] O. Ciupke. *Analysis of Object-Oriented Programs Using Graphs*. In J. Bosch and S. Mitchell, editors, Object-Oriented Technology – Ecoop'97 Workshop Reader, volume 1357 of *Lecture Notes in Computer Science*, pages 270–271, Jyväskylä, Finnland, March 1997. Springer Verlag. (p 213)

[CIUP 99] O. Ciupke. *Automatic Detection of Design Problems in Object-Oriented Reengineering*. In Proceedings of the 30th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA'99). IEEE Computer Society Press, August 1999. (pp 21, 201)

[COAD 91] P. Coad and E. Yourdon. Object-Oriented Analysis. Prentice Hall, London, 2 edition, 1991. (pp 81, 89)

[COLD 99] J. Coldewey, W. Keller, and K. Renzel. Architectural Patterns for Business Information Systems. Publisher Unknown, 1999. (p 127)

[COMM 94] C. T. Committee. *CDIF Framework for Modeling and Extensibility*. Research Report EIA/IS-107, Electronic Industries Association, January 1994. (pp 218, 219)

[CONS 92] M. Consens, A. Mendelzon, and A. Ryman. *Visualizing and Querying Software Structures*. In Proceedings of the 14th International Conference on Software Engineering, pages 138–156, 1992. (p 241)

[COPL 92] J. O. Coplien. Advanced C++: Programming Styles and Idioms. Addison-Wesley, 1992. (pp 9, 111)

[CROS 98] J. H. Cross II, T. D. Hendrix, L. A. Barowsky, and K. S. Mathias. *Scalable Visualizations to Support Reverse Engineering: A Framework for Evaluation*. In Proceedings of WCRE'98, pages 201–210. IEEE Computer Society, 1998. (p 241)

[CZAR 99] K. Czarnecki, U. Gleich, and C. Riva. *Experience Report on the Integrated Re-Engineering Technology*. Achievement A4.5.1, DaimlerChrysler Research and Technology, Nokia Research Center, 1999. (p 254)

[DAVI 95] A. M. Davis. 201 Principles of Software Development. McGraw-Hill, 1995. (pp 8, 121)

[DEMA 82] T. DeMarco. Controlling Software Projects; Management, Measurement and Estimation. Yourdan Press, New Jersey, 1982. (p 261)

[DEME 98a] S. Demeyer and H. Gall. *Workshop on Object-Oriented Re-engineering (WOOR'97)*. Software Engineering Notes, vol. 23, no. 1, pages 28–29, January 1998. (p 141)

[DEME 98b] S. Demeyer, S. Tichelaar, and P. Steyaert. *Definition of a Common Exchange Model*. technical report, University of Berne, July 1998. (p 218)

[DEME 98c] S. Demeyer, S. Tichelaar, and P. Steyaert. *Definition of a Common Exchange Model*. Deliverable A2.4.1, FAMOOS, 1998. (p 250)

[DEME 99a] S. Demeyer and S. Ducasse. *Metrics, Do They Really Help ?* In J. Malenfant, editor, Proceedings LMO'99 (Languages et Modèles à Objets), pages 69–82. HERMES Science Publications, Paris, 1999. (pp 21, 29, 241)

[DEME 99b] S. Demeyer, S. Ducasse, and M. Lanza. *A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization*. In F. Balmas, M. Blaha, and S. Rugaber, editors, WCRE'99 Proceedings (6th Working Conference on Reverse Engineering). IEEE, October 1999. (p 21)

[DEME 99c] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Finding Refactorings via Change Metrics*. working paper, April 1999. (pp 29, 137)

[DEME 99d] S. Demeyer, S. Ducasse, and S. Tichelaar. *Why Unified is not Universal. UML Shortcomings for Coping with Round-trip Engineering*. In B. Rumpe, editor, Proceedings UML'99 (The Second International Conference on The Unified Modeling Language), LNCS ????, Kaiserslautern, Germany, October 1999. Springer-Verlag. (p 218)

[DICK 96] H. Dicky, C. Dony, M. Huchard, and T. Libourel. *On Automatic Class Insertion with Overloading*. In SIGPLAN Notices, vol. 31, no 10 (special issue on OOPSLA'96), pages 251–267, October 1996. (p 316)

[DUMK 97] R. Dumke and E. Foltin. *Matrics-based Evaluation of Object-Oriented Software Development*. SMLAB Report, August 1997. (p 261)

[DYSO 96] P. Dyson and B. Anderson. *State Patterns*. In First European Conference on Pattern Languages of Programming, 1996. (p 196)

[ENGE 98] R. L. Engelbrecht and D. G. Kourie. *Issues in Translating Smalltalk to Java*. In K. Koskimies, editor, Compiler Construction 98, volume 1383 of *LNCS*. Springer, 1998. (p 210)

[ERNI 96] K. Erni and C. Lewerentz. *Applying Design-Metrics to Object-Oriented Frameworks*. In Proceedings of the 3rd International Software Metrics Symposium. IEEE Computer Society Press, 1996. (pp 29, 250)

[ETZK 98] L. Etzkorn, C. Davis, and W. Li. *A Practical Look at the Lack of Cohesion in Methods Metric*. Journal of Object-Oriented Programming, vol. 11, no. 5, pages 27–34, September 1998. (p 25)

[ETZK 99] L. Etzkorn, J. Bansiya, and C. Davis. *Design and Code Complexity Metrics for OO Classes*. Journal of Object-Oriented Programming, pages 35–40, 1999. (p 23)

[FENT 94] N. Fenton. *Software Measurement; A Necessary Scientific Base*. IEEE Transactions on Software Engineering, vol. 20, no. 3, March 1994. (pp 262, 266)

[FENT 97] N. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, Second edition, 1997. (pp 8, 131, 266, 267, 300, 313)

[FIOR 98a] F. Fioravanti, P. Nesi, and S. Perli. *Assessment of System Evolution Through Characterization*. In ICSE'98 Proceedings (International Conference on Software Engineering). IEEE Computer Society, 1998. (p 133)

[FIOR 98b] F. Fioravanti, P. Nesi, and S. Perli. *A Tool for Process and Product Assessment of C++ Applications*. In CSMR'98 Proceedings (Euromicro Conference on Software Maintenance and Reengineering). IEEE Computer Society, 1998. (p 133)

[FLOR 97] G. Florijn, M. Meijers, and P. van Winsen. *Tool Support for Object-Oriented Patterns*. In M. Aksit and S. Matsuoka, editors, Proceedings ECOOP'97, LNCS 1241, pages 472–495, Jyvaskyla, Finland, June 1997. Springer-Verlag. (pp 9, 100)

[FOOT 97] B. Foote and J. W. Yoder. *Big Ball of Mud*. In Proceedings of PLop'97, 1997. (p 163)

[FOWL 97a] M. Fowler. UML Distilled. Addison-Wesley, 1997. (p 9)

[FOWL 97b] M. Fowler. Analysis Patterns: Reusable Objects Models. Addison-Wesley, 1997. (p 124)

[FOWL 99] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999. (pp 10, 21, 111, 134, 139)

[GAMM 91] E. Gamma. *Objektorientierte Software-Entwicklung am Beispiel von ET++: Klassenbibliothek, Werkzeuge, Design*. PhD thesis, 1991. (p 91)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison Wesley, Reading, MA, 1995. (pp 9, 15, 91, 112, 124, 140, 163, 164, 175, 176, 181, 192, 194, 196, 199, 201, 315)

[GOLD 95] A. Goldberg and K. S. Rubin. Succeeding With Objects: Decision Frameworks for Project Management. Addison-Wesley, Reading, Mass., 1995. (pp 8, 105)

[GROT 97] T. Grothehen and K. R. Dittrich. *The MeTHOOD Approach: Measures, Transformation Rules and Heuristics for Object-Oriented Design.* http://www.ifi.unizh.ch/dbtg/MeTHOOD/index.html, October 1997. (p 100)

[HAIN 96] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland. *Database reverse Engineering: From requirements to CARE Tools*. In Automated Software Engineering, Vol. 3 Nos 1/2, June 1996. 1996. (pp 127, 237)

[HALS 77] M. H. Halstead. Elements of Software Science. Elsevier North-Holland, New York, 1977. (p 261)

[HARA 69] F. Harary. Graph Theory. Series in Mathematics. Addison-Wesley, 1969. (p 84)

[HARR 98] R. Harrison, S. Counsell, and R. Nithi. *An Evaluation of the MOOD Set of Object-Oriented Software Metrics.* IEEE Transactions on Software Engineering, vol. 24, no. 6, December 1998. (p 267)

[HELF 95] J. Helfman. *Dotplot Patterns: a Literal Look at Pattern Languages*. TAPOS, vol. 2, no. 1, pages 31–41, 1995. (pp 186, 187, 235, 236)

[HEND 96] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1996. (pp 8, 24, 266)

[HENR 81] S. Henry and D. Kafura. *Software Structure Metrics Based on Information Flow*. IEEE Transactions on Software Engineering, vol. 7, no. 5, pages 510–518, September 1981. (p 261)

[HITZ 95] M. Hitz and B. Montazeri. *Measure Coupling and Cohesion in Object-Oriented Systems*. Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95), October 1995. (pp 24, 25, 274, 279)

[HITZ 96a] M. Hitz and B. Montazeri. *Chidamber and Kemerer's Metrics Suite; A Measurement Theory Perspective*. IEEE Transactions on Software Engineering, vol. 22, no. 4, pages 267–271, April 1996. (pp 24, 25, 279)

[HITZ 96b] M. Hitz and B. Montazeri. *Measuring Coupling in Object-Oriented Systems.* Object Currents, vol. 1, no. 4, 1996. (pp 272, 274, 276, 288)

[HOND 98] K. D. Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. Ph.D. thesis, Vrije Universiteit Brussel - Departement of Computer Science - Pleinlaan 2, Brussels - Belgium, December 1998. (p 137)

[HOSM 89] D. Hosmer and S. Lemeshow. Applied Logistic Regression. Wiley-Interscience, 1989. (p 284)

[HÜ 96] W. Hürsch and L. Seiter. *Automating the Evolution of Object-Oriented Systems*. In Proceedings of ISOTAS'96, LNCS 1049, pages 2–21, Japan, March 1996. JSSST-JAIST. (pp 100, 316)

[HUMP 97] W. Humphrey. Introduction to the Personal Software Process. SEI Series in Software Engineering. Addison Wesley, 1997. (p 22)

[IEEE 83] IEEE. *IEEE Standard*. Research Report 729-1983, IEEE, 1983. (p 317)

[JACO 92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley/ACM Press, Reading, Mass., 1992. (p 124)

[JACO 97] I. Jacobson, M. Griss, and P. Jonsson. Software Reuse. Addison-Wesley/ACM Press, 1997. (pp 105, 114)

[JAHN 97] J. H. Jahnke, W. Schäfer, and A. Zündorf. *Generic Fuzzy Reasoning Nets as a Basis for Reverse Engineering Relational Database Applications*. In Proceedings of ESEC/FSE'97, number 1301 in LNCS, pages 193–210, 1997. (p 127)

[JAZA 99] M. Jazayeri, H. Gall, and C. Riva. *Visualizing Software Release Histories: The Use of Color and Third Dimension*. In ICSM'99 Proceedings (International Conference on Software Maintenance). IEEE Computer Society, 1999. (p 137)

[JERD 97] D. Jerding and S. Rugaber. *Using Visualization for Architectural Localization and Extraction*. In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings Fourth Working Conference on Reverse Engineering, pages 56 – 65. IEEE Computer Society, 1997. (p 241)

[JOHN 88] R. E. Johnson and B. Foote. *Designing Reusable Classes.* Journal of Object-Oriented Programming, vol. 1, no. 2, pages 22–35, 1988. (p 315)

[JOHN 92] R. E. Johnson. *Documenting Frameworks using Patterns.* In Proceedings OOPSLA '92 ACM SIGPLAN Notices, pages 63–76, October 1992. (pp 9, 246)

[JOHN 93] R. E. Johnson and W. F. Opdyke. *Refactoring and Aggregation*. In Object Technologies for Advanced Software, First JSSST International Symposium, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, November 1993. (pp 10, 21, 91, 94, 95, 96, 97, 100, 164, 181, 200, 316)

[KELL 98] W. Keller and J. Coldewey. *Accessing Relational Databases: A Pattern Language*. In R. Martin, D. Riehle, and F. Bushmann, editors, Pattern Languages of Program Design 3, pages 313–343. Addison-Wesley, 1998. (p 127)

[KIME 94] D. Kimelman, B. Leban, T. Roth, and D. Zernik. *Reduction of Visual Complexity in Dynamic Graphs*. In R. Tamassia and I. G. Tollis, editors, Graph Drawing, volume 894 of *Lecture Notes in Computer Science*, pages 218–225. DIMACS, Springer-Verlag, October 1994. (p 83)

[KITC 95] B. Kitchenham, S. Pfleeger, and N. Fenton. *Towards a Framework for Software Measurement Validation*. IEEE Transactions on Software Engineering, vol. 21, no. 12, December 1995. (pp 266, 267, 313)

[KLEY 88] M. F. Kleyn and P. C. Gingrich. *GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views*. In Proceedings OOPSLA '88, ACM SIGPLAN Notices, pages 191–205, November 1988. (p 241)

[KONT 97] K. Kontogiannis. *Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics*. In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings Fourth Working Conference on Reverse Engineering, pages 44 – 54. IEEE Computer Society, 1997. (pp 133, 241)

[KOSK 96] K. Koskimies and H. Mössenböck. *Scene: Using Scenario Diagrams And Active Text for Illustrating Object-Oriented Programs*. In Proceedings of the 18th International Conference on Software Engineering, pages 366–375. IEEE Computer Society Press, March 1996. (p 83)

[KOSK 98a] K. Koskimies. *Extracting High-Level Views of UML Class Diagrams.* In Proceedings of the Nordic Workshop on Software Architecture, NOSA'98, number 14/98 in Research Report. Department of Computer Science, University of Karlskrona/Ronneby, August 1998. (p 88)

[KOSK 98b] K. Koskimies. *TDE / CAN - A C++ Reverse Engineering Tool*. Achievement A2.7.2.1, Nokia Research Center, 1998. (p 251)

[LAGU 97] B. Lague, D. Proulx, E. Merlo, J. Mayrand, and J. Hudepohl. *Assessing the Benefits of Incorporating Function Clone Detection in a Development Process*. In Proceedings of ICSM'97. IEEE, 1997. (p 187)

[LAKO 96] J. Lakos. Large Scale C++ Software Design. Addison-Wesley, 1996. (p 9)

[LAMP 95] J. Lamping, R. Rao, and P. Pirolli. *A Focus + Context Technique Based on Hyperbolic Geometry for Visualising Larges Hierarchies*. In Proceedings of CHI'95, 1995. (pp 31, 241)

[LANG 95] D. B. Lange and Y. Nakamura. *Interactive Visualization of Design Patterns can help in Framework Understanding*. In Proceedings of OOPSLA'95, pages 342–357. ACM Press, 1995. (p 237)

[LANZ 99] M. Lanza. Combining Metrics And Graphs for Object Oriented Reverse Engineering. Master's thesis, University of Bern, 1999. (pp 21, 246)

[LEA 96] D. Lea. Concurrent Programming in Java – Design principles and Patterns. The Java Series. Addison-Wesley, 1996. (p 124)

[LERN 90] B. S. Lerner and A. N. Habermann. *Beyond Schema Evolution to Database Reorganization*. In Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, pages 67–76, October 1990. (p 315)

[LEWE 98a] C. Lewerentz and F. Simon. *A Product Metrics Tool Integrated into a Software Development Environment*. In Object-Oriented Technology Ecoop'98 Workshop Reader, LNCS 1543, pages 256–257, 1998. (pp 133, 241)

[LEWE 98b] C. Lewerentz and F. Simon. *A Product Metrics Tool Integrated into a Software Development Environment*. In Proceedings of the European Software Measurement Conference (FESMA '98), Belgium, 1998. (p 253)

[LI 93] W. Li and S. Henry. *Maintenance Metrics for the Object Oriented Paradigm*. IEEE Proceedings of the First International Software Metrics Symposium, pages 52–60, May 1993. (pp 24, 25, 270, 272, 273, 274, 279)

[LI 96] Q. Li and D. McLeod. *Object Flavor Evolution through Learning in an Object-Oriented Database System*. In Proceedings of the 2nd International Conference on Expert Database Systems, pages 241–256. George Mason University, April 1996. (p 317)

[LI 98] J. Li. *Maintenance Support for Untyped Object-Oriented Systems.* http://www.informatik.uni-stuttgart.de/ifi/se/people/li/, 1998. (pp 213, 214)

[LIEB 88] K. J. Lieberherr, I. M. Holland, and A. Riel. *Object-Oriented Programming: An Objective Sense of Style*. In Proceedings OOPSLA '88, ACM SIGPLAN Notices, pages 323–334, November 1988. (p 316)

[LIEB 89] K. Lieberherr and I. Holland. *Assuring a Good Style for Object-Oriented Programs*. IEEE Software, pages 38–48, September 1989. (p 66)

[LIEB 91] K. Lieberherr, P. Bergstein, and I. Silva-Lepe. *From Objects to Classes: Algorithms for Optimal Object-Oriented Design*. Software Engineering Journal, pages 205–228, July 1991. (p 315)

[LIEB 95] K. J. Lieberherr. Adaptive Object-Oriented Software – The Demeter Method. PWS Publishing Company, 1995. (p 100)

[LORE 94] M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Approach. Prentice-Hall, 1994. (pp 8, 23, 24, 29, 241)

[MARI 97] R. Marinescu. *The Use of Software Metrics in the Design of Object-Oriented Systems*. Diploma thesis, University Politehnica Timisoara - Fakultat fur Informatik, October 1997. (pp 213, 262)

[MARI 98] R. Marinescu. *Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems.* In Object-Oriented Technology Ecoop'98 Workshop Reader, LNCS 1543, pages 252–253, 1998. (pp 133, 241)

[MARI 99] R. Marinescu. *A Multi-Layered System of Metrics for the Measurement of Reuse by Inheritance.* In Proc. of the 31st TOOLS'99 Conference. IEEE Computer Press, 1999. (p 304)

[MAYR 96a] J. Mayrand, C. Leblanc, and E. Merlo. *Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics*. In International Conference on Software System Using Metrics, pages 244–253, 1996. (p 133)

[MAYR 96b] J. Mayrand, C. Leblanc, and E. M. Merlo. *Automatic detection of Function Clones in a Software System using Metrics.* In Proceedings of ICSM (International (International Conference on Software Maintenance), 1996. (pp 186, 237)

[MCCA 76] T. J. McCabe. *A Complexity Measure*. IEEE Transactions on Software Engineering, vol. 2, no. 4, pages 308–320, December 1976. (p 261)

[MCCA 77] J. McCall. *Factors in Software Quality*. RADC TR-77-369, vol. I, II, III, 1977. (p 301)

[MCCL 78] C. L. McClure. *A Model for Program Complexity Analysis*. Proceeding of the 3rd International Conference on Software Engineering, pages 149–157, May 1978. (p 261)

[MEIJ 96] M. Meijers. Tool Support for Object-Oriented Design Patterns. Master's thesis, CS Department of Utrecht University, August 1996. (p 100)

[MEYE 96] S. Meyers. More Effective C++. Addison-Wesley, 1996. (pp 9, 111)

[MEYE 97] B. Meyer. Object-Oriented Software Construction. Prentice Hall, Second edition, 1997. (pp 9, 141, 143)

[MEYE 98] S. Meyers. Effective C++. Addison-Wesley, second edition, 1998. (pp 9, 111)

[MOHR 98] B. Mohr. Reorganisation objektorientierter Systeme. Diplomarbeit, Forschungszentrum Informatik (FZI) an der Universität Karlsruhe (TH), Karlsruhe, Germany, March 1998. (pp 95, 96, 98, 201)

[MOOR 96] I. Moore. *Automatic Inheritance Hierarchy Restructuring and Method Refactoring*. In Proceedings of OOPSLA '96 Conference, pages 235–250. ACM Press, 1996. (p 10)

[MÜ 86] H. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986. (p 241)

[MURP 97] G. Murphy and D. Notkin. *Reengineering with Reflexion Models: A Case Study*. IEEE Computer, vol. 17, no. 2, pages 29–36, August 1997. (p 125)

[NESI 88] P. Nesi. *Managing OO Project Better*. IEEE Software, July 1988. (p 133)

[ODEN 97] G. Odenthal and K. Quibeldey-Cirkel. *Using Patterns for Design and Documentation*. In Proceedings of ECOOP'97, LNCS 1241, pages 511–529. Springer-Verlag, June 1997. (p 9)

[OPDY 90] W. F. Opdyke and R. E. Johnson. *Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems*. In Proceedings of SOOPPA, ACM, November 1990, pages 145–160, 1990. (p 316)

[OPDY 92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. (pp 10, 201)

[OPDY 93] W. F. Opdyke and R. E. Johnson. *Creating Abstract Superclasses by Refactoring*. In Proceedings of the 1993 ACM Conference on Computer Science, pages 66–73. ACM Press, 1993. (p 10)

[ORGA 91] I. S. Organization. *Information Technology – Software Product Evaluation – Quality Characteristics and Guide Lines for their Use*, 1991. (p 301)

[OXHØ 92] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. *Making Type Inference Practical*. In O. L. Madsen, editor, Proceedings ECOOP'92, LNCS 615, pages 329–349, Utrecht, The Netherlands, June 1992. Springer-Verlag. (p 212)

[PALS 91] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Inference*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, pages 146–161, November 1991. (p 212)

[PAUW 93] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proceedings OOPSLA '93, ACM SIGPLAN Notices, pages 326–337, October 1993. (pp 31, 83, 241, 317)

[PLEV 94] J. Plevyak and A. A. Chien. *Precise Concrete Type Inference for Object-Oriented Languages*. ACM SIGPLAN Notices, vol. 29, no. 10, pages 324–324, October 1994. (p 212)

[PREC 98] L. Prechelt and C. Krämer. *Functionality versus Practicality: Employing Existing Tools for Recovering Structural Design Patterns*. Journal of Universal Computer Science, vol. 4, no. 12, pages 866–882, December 1998. (p 9)

[PREM 94] W. J. Premerlani and M. R. Blaha. *An Approach for Reverse Engineering of Relational Databases*. Communications of the ACM, vol. 37, no. 5, pages 42–49, May 1994. (p 127)

[PRES 94] R. S. Pressman. Software Engineering: A Practitioner's Approach. McGraw-Hill, 1994. (p 8)

[PUST 82] J. Pustell and F. Kafatos. *A High Speed, High Capacity Homology Matrix: Zooming through sv40 and Polyoma*. Nucleid Acids Research, vol. 10, no. 15, pages 4765–4782, 1982. (p 187)

[PUTK 94] A. Putkonen. *A Methodology Supporting Analysis, Design and Maintenance of Object-Oriented Systems*. PhD thesis, University of Kuopio, Finland, 1994. (p 315)

[RA 95] Y.-G. Ra and E. A. Rundensteiner. *A Transparent Object-Oriented Schema Change Approach Using View Evolution*. In Proceedings of the 11th International Conference on Data Engineering, pages 165–172. IEEE Computer Society Press, March 1995. (p 316)

[RANS 98] J. Ransom, I. Sommerville, and I. Warren. *A Method for Assessing Legacy Systems for Evolution*. In Proceedings of Reengineering Forum'98, 1998. (p 164)

[RAPI 98] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. *Dynamic Type Inference to Support Object-Oriented Reengineering in Smalltalk*, 1998. (p 211)

[REEN 96] T. Reenskaug. Working with Objects: The OOram Software Engineering Method. Manning, 1996. (pp 105, 124)

[RIEG 98] M. Rieger and S. Ducasse. *Visual Detection of Duplicated Code*. In S. Demeyer and J. Bosch, editors, Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543, pages 75–76. Springer-Verlag, July 1998. (p 37)

[RIEL 96] A. J. Riel. Object-Oriented Design Heuristics. Addison-Wesley, May 1996. (pp 9, 28, 229, 231)

[RITZ 98] F. Ritzmann. Reverse Engineering of Large Scale Software Systems. Diplomarbeit, Universität Karlsruhe, June 1998. (pp 88, 208)

[RIVA 98] C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. Master's thesis, Politecnico di Milano, Milan, 1998. (p 31)

[RIVA 99] C. Riva, M. Przybilski, and K. Koskimies. *Environment for Software Assessment.* In Workshop on Object-Oriented Architectural Evolution, 13th European Conference on Object-Oriented Programming (ECOOP 99), Lisabon, Portugal, June 1999. (p 249)

[ROBE 96] D. Roberts and R. Johnson. *Evolving Frameworks - A Pattern Language for Developing Object -Oriented Frameworks*. http://st-www.cs.uiuc.edu/users/droberts/evolve.html, 1996. (p 202)

[ROBE 97a] D. Roberts, J. Brant, and R. E. Johnson. *A Refactoring Tool for Smalltalk.* Journal of Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997. (pp 10, 21, 37, 113, 201, 209, 243)

[ROBE 97b] D. Roberts, J. Brant, and R. Johnson. *A Refactoring Tool for Smalltalk*. http://st-www.cs.uiuc.edu/users/brant/Refactory/Refac toringBrowser.html, April 1997. (pp 95, 96, 100)

[ROBI 89] P. Robillard and G. Boloix. *The Interconnectivity Metrics; A New Metric Showing How a Program is Organised*. Journal of Systems and Software, vol. 10, pages 29–39, October 1989. (p 261)

[ROCH 93] R. Rochat and J. Ewing. *Smalltalk Debugging Techniques*. The Smalltalk Report, vol. 2, no. 9, pages 18–23, jul 1993. (p 143)

[RUMB 99] J. Rumbaugh, I. Jacobson, and G. Booch. The Unified Modeling Language Reference. Addison-Wesley, 1999. (pp 9, 81)

[SAND 96] G. Sander. *Graph Layout for Applications in Compiler Construction*. Research report, Universitaet des Saarlandes, February 1996. (p 241)

[SCHM 89] G. Schmidt and T. Ströhlein. Relationen und Graphen. Mathematik für Informatiker. Springer-Verlag, 1989. (p 84)

[SCHU 98a] B. Schulz, T. Genssler, B. Moor, and W. Zimmer. *On the Computer Aided Introduction of DEsign Patterns into Object-Oriented Systems.* In Proceedings of the 27th TOOLS conference, IEEE CS Press, 1998. (p 10)

[SCHU 98b] B. Schulz, T. Genssler, B. Mohr, and W. Zimmer. *On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems.* In 27th Conference on Technology of Object-Oriented Languages and Systems (TOOLS). IEEE, 1998. (p 200)

[SEFI 96] M. Sefika, A. Sane, and R. H. Campbell. *Architecture-Oriented Visualization*. In SIGPLAN Notices, vol. 31, no 10 (special issue on OOPSLA'96), pages 389–407, October 1996. (p 317)

[SKAR 86] A. H. Skarra and S. B. Zdonik. *The Management of Changing Types in an Object-Oriented Database*. In Proceedings OOPSLA '86, ACM SIGPLAN Notices, pages 483–495, November 1986. (p 315)

[SOFT 97] R. Software, Microsoft, Hewlett-Packard, Oracle, S. Software, M. Systemhouse, Unisys, I. Computing, IntelliCorp, i Logix, IBM, ObjecTime, P. Technology, Ptech, Taskon, R. Technologies, and Softeam. Unified Modeling Language (version 1.1). Rational Software Corporation, September 1997. (p 165)

[SOMM 96] I. Sommerville. Software Engineering. Addison-Wesley, Fifth edition, 1996. (p 8)

[SPIV 92] J. M. Spivey. The Z Notation. Prentice Hall, New York, second edition, 1992. (p 85)

[STEV 98] P. Stevens and R. Pooley. *System Reengineering Patterns*. In Proceedings of FSE-6. ACM-SIGSOFT, 1998. (pp 163, 164)

[STEY 96] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. *Reuse Contracts: Managing the Evolution of Reusable Assets*. In Proceedings of OOPSLA '96 Conference, pages 268–285. ACM Press, 1996. (p 9)

[STIG 95] B. Stiglic, M. Hericko, and I. Rozman. *How to Evaluate Object-Oriented Software Development?* ACM SIGPLAN Notices, vol. 30, no. 5, May 1995. (p 272)

[STOR 95] M.-A. D. Storey and H. A. Müller. *Manipulating and documenting software structures using SHriMP views*. In Proceedings of the 1995 International Conference on Software Maintenance, 1995. (p 241)

[SUGI 81] K. Sugiyama, S. Tagawa, and M. Toda. *Methods for Visual Understanding of Hierarchical System Structures*. IEEE Transactions on Systems, Man and Cybernetics, vol. SMC-11, no. 2, February 1981. (p 241)

[TAKE 99] I. TakeFive Software. *Sniff+*, 1999. (p 251)

[TEGA 95] T. Tegarden, S. Sheetz, and D. Monarchi. *A Software Complexity Model of Object-Oriented Systems*. Decision Support Systems, vol. 13, no. 3–4, pages 241–262, 1995. (p 26)

[TICH 98] S. Tichelaar and S. Demeyer. *An Exchange Model for Reengineering Tools*. In S. Demeyer and J. Bosch, editors, Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543. Springer-Verlag, July 1998. (p 246)

[TICH 99] S. Tichelaar and S. Demeyer. *SNiFF+ Talks to Rational Rose -Interoperability using a Common Exchange Model*. In SNiFF+ User's Conference, January 1999. (p 251)

[VAN 96] P. van Winsen. Reengineering with Object-Oriented Design Patterns. Master's thesis, Cs Department of Utrecht University, November 1996. (p 100)

[WATE 94] R. C. Waters and E. Chikofsky. *Reverse Engineering: Progress Along Many Dimensions (Special Issue)*. Communications of the ACM, vol. 37, no. 5, pages 22–93, May 1994. (p 8)

[WEIN 89] A. Weinand, E. Gamma, and R. Marty. *Design and Implementation of ET++, a Seamless Object-Oriented Application Framework*. Structured Programming, vol. 10, no. 2, pages 63–87, 1989. (p 231)

[WEYU 88] E. J. Weyuker. *Evaluating Software Complexity Measures*. IEEE Transactions on Software Engineering, vol. 14, no. 9, September 1988. (p 266)

[WHIT 97] S. A. Whitmire. Object-Oriented Design Measurement. Wiley, September, 1997. (p 300)

[WILD 92] N. Wilde and R. Huit. *Maintenance Support for Object-Oriented Programs.* Transactions on Software Engineering, vol. 18, no. 12, pages 1038–1044, December 1992. (p 11)

[WILL 96a] L. Wills and P. Newcomb. *Reverse Engineering (Special Issue)*. Automated Software Engineering, vol. 3, no. 1-2, pages 5–172, June 1996. (p 8)

[WILL 96b] L. Wills and J. H. Cross, II. *Recent Trends and Open Issues in Reverse Engineering*. Automated Software Engineering, vol. 3, no. 1-2, pages 165–172, June 1996. (p 107)

[WINS 87] M. E. Winston, R. Chaffin, and D. Herrmann. *A Taxonomy of Part-Whole Relations*. Cognitive Science, vol. 11, pages 417–444, 1987. (p 9)

[WIRF 90] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. Designing Object-Oriented Software. Prentice Hall, 1990. (p 124)

[WOOD 80] S. N. Woodfield. *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*. PhD thesis, Computer Science Deparment, Purdue University, December 1980. (p 261)

[WUYT 98] R. Wuyts. *Declarative Reasoning about the Structure of Object-Oriented Systems.* In Proceedings of TOOLS USA '98. IEEE, August 1998. (p 9)

[YELL 96] P. Yelland. *Creating Host Compliance in a Portable Framework: A Study in the Use of Existing Design Patterns.* In SIGPLAN Notices, vol. 31, no 10 (special issue on OOPSLA'96), pages 18–29, October 1996. (p 316)

[ZICA 92] Zicari. A Framework for Schema Updates in an Object-Oriented Database System, in Building an Object-Oriented Database System - The Story of O2. Morgan Kaufmann Publishers, 1992. (p 317)

[ZIMM 95] W. Zimmer. *Using Design Patterns to Reorganize Object-Oriented Applications.* Research Report 1/95, FZI, 1995. (pp 91, 197, 198, 201, 316)

[ZIMM 97] W. Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, 1997. (pp 91, 92, 96, 97, 200)

[ZUSE 90] H. Zuse. Software Complexity: Measures and Methods. Walter de Gruyter, Berlin, 1990. (pp 266, 267)

[ZUSE 95] H. Zuse. *Properties of Object-Oriented Software Measures*. 5th Worshop of GI Berlin Arbeitsgruppe Softwaremetriken, September 1995. (p 262)

[Sem 97] SemaGroup. *FAST Programmer's Manual and FAST Programmer's Manual Complement*, 1997. (p 98)

[Sem 99] Sema Group, http://www.sema.com. *Concerto2 / Audit*, 1999. (p 252)

[STS 97] *Software Reengineering Assessment Handbook v3.0.* Research report, STSC, U.S. Department of Defense, March 1997. (p 164)