Object-Oriented Reverse Engineering

Coarse-grained, Fine-grained, and Evolutionary Software Visualization

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michele Lanza

von Italien

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Object-Oriented Reverse Engineering

Coarse-grained, Fine-grained, and Evolutionary Software Visualization

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michele Lanza

von Italien

Leiter der Arbeit: Prof. Dr. S. Ducasse, Prof. Dr. O. Nierstrasz
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 06.05.2003

Der Dekan:

Prof. G. Jäger

Abstract

The maintenance, reengineering, and evolution of object-oriented software systems has become a vital matter in today's software industry. Although most systems start off in a clean and well-designed state, with time they tend to gradually decay in quality, unless the systems are reengineered and adapted to the evolving requirements. However, before such *legacy* software systems can be reengineered and evolved, they must be reverse engineered, *i.e.*, their structure and inner working must be understood. This is difficult because of several factors, such as the sheer size of the systems, their complexity, their domain specificity, and in general the bad state legacy software systems are in.

In this thesis we propose a visual approach to the reverse engineering of object-oriented software systems by means of *polymetric views*, lightweight visualizations of software enriched with metrics and other types of semantic information about the software, *e.g.*, its age, version, abstractness, location, structure, function, etc.

We present and discuss several polymetric views which allow us to understand three different aspects of object-oriented software, namely

1. coarse-grained aspects which allow for the understanding of very large systems,
2. fine-grained aspects which allow for the understanding of classes and class hierarchies,
3. and evolutionary aspects, which enable us to recover and understand the evolution of a software system.

The combination of these three types of information can greatly reduce the time needed to gain an understanding of an object-oriented software system.

Based on the application of our polymetric views, we present our reverse engineering methodology which we validated and refined on several occasions in industrial settings. It allows us to explore and combine these three approaches into one single visual approach to *understand software*.

Contents

Abstract **i**

1 Introduction **1**

1.1 The Problem . . . 1
1.2 A Short Reverse Engineering State-of-the-Art . . . 2
1.3 Our Approach . . . 3
1.4 Contributions . . . 5
1.5 Thesis Outline . . . 5

2 Object-Oriented Reverse Engineering **7**

2.1 Introduction . . . 7
2.2 Object-Oriented Reverse Engineering . . . 8
2.2.1 General Approaches to Reverse Engineering . . . 8
2.2.2 Challenges and Goals in Object-Oriented Reverse Engineering . . . 10
2.3 Our Work: Scope, Constraints, and Goals . . . 11
2.3.1 Scope . . . 11
2.3.2 Constraints . . . 13
2.3.3 Goals . . . 13
2.4 Conclusions . . . 14

3 The Polymetric View **15**

3.1 The Principle . . . 15
3.2 Software Metrics . . . 16
3.3 The Actual Visualization: A Polymetric View . . . 18
3.4 Two Example Polymetric Views . . . 19
3.5 Interpretation of a Polymetric View . . . 20
3.6 Useful Polymetric Views . . . 21
3.7 Discussion . . . 21

4 Coarse-grained Software Visualization **22**

4.1 Introduction . . . 22
4.2 The Reverse Engineering Process . . . 23
4.2.1 A Reverse Engineering Approach Based on Clusters of Polymetric Views . . . 23
4.2.2 The Need for a Reverse Engineering Approach . . . 24
4.3 A Reverse Engineering Scenario . . . 25
4.3.1 Reverse Engineering a System . . . 26
4.3.2 First Contact Views . . . 28
4.3.3 Inheritance Assessment Views . . . 37
4.3.4 Candidate Detection Views . . . 43
4.3.5 Case Study Evaluation . . . 52
4.3.6 Industrial Experiences . . . 52
4.4 Related Work . . . 54

4.5 Conclusion . . . 55
4.5.1 Summary . . . 55
4.5.2 Benefits . . . 55
4.5.3 Limits . . . 56
4.5.4 Future Work . . . 56

5 Fine-grained Software Visualization: The Class Blueprint 57
5.1 Introduction . . . 57
5.2 The Challenge of Supporting Class Understanding . . . 58
5.3 The Class Blueprint . . . 60
5.3.1 The Layered Structure of a Class Blueprint . . . 60
5.3.2 Representing Methods and Attributes . . . 61
5.3.3 The Layout Algorithm of a Class Blueprint . . . 63
5.4 Detailing Blueprints . . . 64
5.4.1 Example 1: An Abstract Class . . . 64
5.4.2 Example 2: An Algorithm . . . 66
5.4.3 Class Blueprints and Inheritance . . . 67
5.5 A Vocabulary based on Patterns in the Class Blueprints . . . 69
5.6 Single Class Blueprint Patterns . . . 70
5.6.1 Size-based Blueprint Patterns . . . 71
5.6.2 Layer Distribution-based Blueprint Patterns . . . 74
5.6.3 Semantics-based Blueprint Patterns . . . 76
5.6.4 Call-flow-based Blueprint Patterns . . . 80
5.6.5 State Usage-based Blueprint Patterns . . . 84
5.7 Blueprint Patterns in the Context of Inheritance . . . 86
5.8 Analysis of Blueprint Patterns . . . 89
5.8.1 Frequent Blueprint Pattern Combinations . . . 89
5.8.2 Rare Blueprint Pattern Combinations . . . 90
5.9 Tool Support: CodeCrawler and Moose . . . 91
5.10 Related Work . . . 92
5.11 Conclusion . . . 92
5.11.1 Summary . . . 92
5.11.2 Benefits . . . 92
5.11.3 Limits . . . 93
5.11.4 Future work . . . 93

6 Evolutionary Software Visualization: The Evolution Matrix 95
6.1 Introduction . . . 95
6.2 The Evolution Matrix . . . 96
6.2.1 The Layout Algorithm of the Evolution Matrix . . . 96
6.2.2 Characteristics at System Level . . . 97
6.2.3 The Difference Evolution Matrix . . . 98
6.3 A Categorization of Classes based on the Evolution Matrix . . . 98
6.3.1 Presence-based Patterns . . . 98
6.3.2 Shape-based Patterns . . . 99
6.4 Illustration of the Approach . . . 101
6.4.1 MooseFinder . . . 101
6.4.2 Supremo . . . 101
6.5 Tool Support: CodEVolver . . . 102
6.6 Related Work . . . 102
6.7 Conclusion . . . 104
6.7.1 Benefits . . . 104
6.7.2 Limits . . . 105
6.7.3 Future Work . . . 106

7 Conclusions **107**
7.1 Contributions 107
7.2 Future Work 108
7.3 Closing Words 110

8 Epilogue **111**

A CodeCrawler - Implementation **112**
A.1 Introduction 112
A.2 CodeCrawler 113
A.2.1 Overall Architecture 114
A.2.2 Internal Architecture 115
A.2.3 Visualization Engine 117
A.2.4 Metamodel 117
A.2.5 Interactive Facilities 118
A.3 Lessons Learned 119
A.3.1 Overall architecture 120
A.3.2 The Internal Architecture 121
A.3.3 The Visualization Engine 122
A.3.4 The Metamodel 122
A.3.5 The Interactive Facilities 123
A.4 Conclusion 123

List of Figures

1.1 The polymetric view SYSTEM COMPLEXITY applied on CodeCrawler. 4

2.1 The reengineering life-cycle of software. 9

3.1 Up to 5 metrics can be visualized on one node. 15
3.2 The polymetric view SYSTEM COMPLEXITY. 19
3.3 The polymetric view METHOD EFFICIENCY CORRELATION. 20

4.1 A schematic view of our reverse engineering approach. 24
4.2 A screenshot from the Duploc case study. 26
4.3 The SYSTEM HOTSPOTS view of Duploc. 30
4.4 The SYSTEM COMPLEXITY view of Duploc. 32
4.5 The ROOT CLASS DETECTION view of Duploc. 33
4.6 The IMPLEMENTATION WEIGHT DISTRIBUTION view of Duploc. 35
4.7 The INHERITANCE CLASSIFICATION view of Duploc. 38
4.8 The INHERITANCE CARRIER view applied on Duploc. 40
4.9 The DATA STORAGE CLASS DETECTION view applied on Duploc. 44
4.10 The METHOD EFFICIENCY CORRELATION view applied on Duploc. 46
4.11 The DIRECT ATTRIBUTE ACCESS view applied on Duploc. 49
4.12 The METHOD LENGTH DISTRIBUTION view applied on one class of Duploc. 50
4.13 The METHOD LENGTH DISTRIBUTION view applied on one class of Duploc. 51
4.14 The SYSTEM HOTSPOTS view applied on 1.2 million lines of code of C++. 53
4.15 The SYSTEM COMPLEXITY view applied on a C++ hierarchy yields a *flying saucer* hierarchy. 54

5.1 A class blueprint decomposes a class into layers. 60
5.2 A graphical representation of methods and attributes using metrics: the metrics are mapped on the width and the height of a node. 62
5.3 The caller has outgoing edges at the bottom, while the callee has in-going edges at the top. 62
5.4 The methods and attributes are positioned according to the layer they have been assigned to. 64
5.5 Left: An blueprint of the class *JunOpenGL3dGraphAbstract*, which represents OpenGL three-dimensional graphs. Right: The same class displayed with method names. 65
5.6 A blueprint of the class *JunSVD*. This class blueprint shows patterns of the type *Single Entry*, *Structured Flow* and *All State*. 66
5.7 A class blueprint visualization of an inheritance hierarchy with the class *JunColorChoice* as root class. The root class contains an *Interface* pattern, while each of the subclasses is a pure *Overrider*. Furthermore, each subclass is a pure *Siamese Twin*. 68
5.8 A class blueprint visualization of an inheritance hierarchy with the class *JunPrologEntity* as root class. 71
5.9 The blueprint of the class *JunSourceCodeSaver* contains a *Large Implementation*, a *Single Entry*, and a *Structured Flow* pattern at the bottom. 72
5.10 The root class is a combination of a nearly pure *Interface* and a *Constant Definer* pattern, while the subclass (*JunAngle*) is a combination of a *Wide Interface* and a *Funnel* pattern. . 74

5.11 The class blueprint of the class *JunOpenGL3dGraphAbstract*: it mainly consists of a *Delegate* pattern. 76
5.12 The class blueprint of the class *JunJfifColorComponent*: it contains a *Data Storage* and a *Three Layers* pattern. 77
5.13 The class blueprint of the class *JunAngleAbstract*: it contains a distinct *Constant Definer* pattern. 78
5.14 A class blueprint visualization of an inheritance hierarchy with the class *JunPrologEntity* as root class. 79
5.15 The class blueprint of the class *JunBmpImageStream* with two *Single Entry* patterns. . . . 80
5.16 The *Structured Flow* pattern of the class *JunSourceCodeSaver*. 81
5.17 The *Method Clumps* pattern of the class *JunGNUDiff*. 82
5.18 A *Funnel* pattern in the class blueprint of the class *JunMovieHandle*. 83
5.19 A *Sharing Entries* pattern in the class blueprint of the class *JunMovieHandle*. 84
5.20 The class blueprint of the class *JunBmpImageStream* with an *All State* pattern. 85
5.21 A class blueprint visualization of an inheritance hierarchy with the class *JunColorChoice* as root class. Each subclass contains a pure *Siamese Twin* pattern. 86
5.22 The subclass in this case shows an *Island* pattern, as it does not communicate at all with its superclass. 87
5.23 A nearly pure *Template* class blueprint of the root class *JunParametricSection*. The two subclasses are both heavy *Overrider* blueprints. 88
5.24 A nearly pure *Interface* class blueprint of the root class *JunGeometry*. The first subclass is a nearly pure *Template* blueprint, while the two leaf classes are combinations of the *Overrider*, and *All State* blueprint. Moreover they are *Siamese Twin* blueprints compared with each other. 90

6.1 A schematic display of the Evolution Matrix. Classes A,D and F are alphabetically ordered and stay since version 1. Classes B and C appeared after version 2. 96
6.2 System level evolution aspects using the Evolution Matrix. 97
6.3 The visualization of *Dayfly* and *Persistent* classes. 99
6.4 The Visualization of a *Pulsar* class.Note that the shape may change depending on the metrics associated with the representation. 99
6.5 The visualization of a *Supernova* class. 100
6.6 The visualization of a *White Dwarf* class. 100
6.7 The visualization of a *Red Giant* class. 101
6.8 The visualization of an *Idle* class. 101
6.9 The Evolution Matrix of MooseFinder. 102
6.10 The Difference Evolution Matrix of MooseFinder. 103
6.11 The Evolution Matrix of Supremo. 104
6.12 The Difference Evolution Matrix of Supremo. 105

A.1 A snapshot of CodeCrawler's main window. The visualized system in this case is CodeCrawler itself. 114
A.2 The general architecture of CodeCrawler. 115
A.3 The core model of CodeCrawler. 116
A.4 CodeCrawler's ViewEditor. 117
A.5 The visualization engine of CodeCrawler. 118
A.6 The architecture of Moose, CodeCrawler's metamodel. 119
A.7 The core of the FAMIX metamodel. 119
A.8 A screenshot of CodeCrawler at work. 120

List of Tables

3.1 An exemplification of our measurement mapping function based on a minimal node size (MNS) of 4. 16
3.2 A list of the class metrics used in this thesis. 17
3.3 A list of the method metrics used in this thesis. 17
3.4 A list of the attribute metrics used in this thesis. 18

4.1 An overview of the size of the Duploc case study. 26
4.2 A list of some of the industrial case studies to which CodeCrawler was applied to. 53

5.1 In a class blueprint semantic information is mapped on the colors of the nodes and edges. . 63

Chapter 1

Introduction

Reverse engineering existing software systems has become an important problem that needs to be tackled. It is defined by Chikosfky and Cross as "the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction" [CHIK 90]. It is the prerequisite for the maintenance, reengineering, and evolution of software systems. Since an unwary modification of one part of a system can have a negative impact, *e.g.*, break, other parts of the system, one needs first to reverse engineer, *e.g.*, have an informed mental model of the software, before the software system can be modified or reengineered.

Sommerville [SOMM 00] and Davis [DAVI 95] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. It would thus seem advisable to rewrite software systems as soon as they fail to fulfill their requirements. However, certain software systems are too valuable to be replaced or to be rewritten, because their sheer size and complexity makes such a feat too expensive for the owning company in terms of time and money. In the case of such *legacy* software systems it is more advisable to first reverse engineer and then maintain, reengineer, and evolve such systems. By adapting them to new requirements [CASA 98, RUGA 98] the lifetime of these systems can be extended and thus increase the return of investment of their owners. Indeed, the longer a software system can be used, the better it pays off for the company that developed it.

We have focused ourselves on the reverse engineering of *object-oriented* legacy systems, mainly because most current software systems are written in languages implementing this paradigm, and because it is not *age* that turns a piece of software into a legacy system, but the *rate* at which it has been developed and adapted [DEME 02]. Moreover, early adopters of object-oriented technology are discovering that the benefits they expected to achieve by switching to objects have been very difficult to realize [DEME 02] and find themselves with present and future legacy systems implemented with object-oriented technology. Moreover, reverse engineering object-oriented software systems comes with additional challenges [WILD 92] compared to non-object-oriented systems, such as polymorphism, late-binding, incremental class definitions, etc.

1.1 The Problem

Reverse engineering software systems, especially large legacy systems, is technically difficult, because they typically suffer from several problems, such as developers no longer available, outdated development methods that have been used to write the software, oudated or completely missing documentation, and in general a progressive degradation of design and quality.

The goal of a person who is reverse engineering a software system is to build progressively refined mental models of the system [STOR 99] to be able to make informed decisions regarding the software. While this is not a complex problem for small software systems, where code reading and inspection is often enough, in the case of legacy software systems which tend to be large – hundreds of thousands or millions of lines of poorly documented code are no exception – this becomes a hard problem because of their sheer size and complexity, and because of the problems afflicting such systems [PARN 94]. In order to build a progressively refined mental model of a software system, the reverse engineer must gather information about the system which helps him in this process.

This leads us to the following research question:

How does a reverse engineer gather the kind of information that he needs in order to build a mental model of an object-oriented software system which allows him to make informed decisions regarding its maintenance, reengineering, and evolution? In other words, *what do we need to know about a software system to understand it?*

We argue that the necessary information resides at various granularity levels, *e.g.*, we need to know and combine information at a coarse-grained level (information about the whole system and its overall structure), at a fine-grained level (information about the structure of classes and class hierarchies), and information at an evolutionary level (information about the evolution of the system and the classes). Our research question can thus be broken down into concrete and simpler research questions, such as:

- How big is the software system and how is it structured?
- What is the architecture of the system and what are the subsystems?
- Where are the most important classes and class hierarchies which represent the problem domain and make the whole software work?
- How are class hierarchies built?
- What is the quality of the software, are all subsystems and/or classes of the same quality, or are there better and worse parts in terms of implementation and reusability?
- What are the design and implementation plans of classes and class hierarchies, *i.e*, how do these look from the inside?
- Where have design patterns [GAMM 95] been used and which ones?
- How did a class or a set of classes or a complete system evolve until its present state?

The goal of this thesis is to provide answers for these and other questions listed in Chapter 2 by using a lightweight approach based on software visualization enriched with software metrics information. We call these visualizations *polymetric views*. The polymetric views we present in this thesis reside at the granularity levels mentioned previously and aim at providing concrete information about a software system useful for its reverse engineering. Moreover, our approach aims at supporting existing reverse engineering approaches and not to replace them.

1.2 A Short Reverse Engineering State-of-the-Art

There are many approaches to reverse engineering software systems, such as:

- reading the existing documentation and source code. This is difficult when the documentation is obsolete, incorrect or not present at all. Reading the source code is a widely used practice, but does not scale up, as reading millions of lines of code would take weeks or months without necessarily increasing the understanding of the system by the reader. Moreover, at the beginning of a reverse engineering process one does not seek detailed information, but rather wants to have a general view of the system.
- running the software and/or generate and analyze execution traces. The use of dynamic information, *e.g.*, information gathered during the execution of a piece of software, has also been used in the context of reverse engineering [RICH 99], but has drawbacks in terms of scalability (traces of a few seconds can become very big) and interpretation (thousands of message invocation can hide the important information one is looking for).

- interviewing the users and developers. This can give important insights into a software system, but is problematic because of the subjective viewpoints of the interviewed people and because it is hard to formalize and reuse these insights.
- using various tools (visualizers, slicers, query engines, etc.) to generate high-level views of the source code. Tool support is provided by the research community in various ways, and visualization tools like Rigi [MÜ 86] and ShrimpViews [STOR 95] are widely used.
- analyzing the version history. Still a young research field, understanding the evolution of a piece of software is done using techniques like graph rewriting, visualization, concept analysis, clustering, and data mining. The insights gained is useful to understand the past of a piece of software and to possibly predict its future.
- assessing a software system and its quality by using software metrics. Software metrics tools are used to assess the quality and quantity of source code by computing various metrics which can be used to detect outliers and other parts of interest, for example cohesive classes, coupled subsystems, etc.

Several of these approaches succeed in tackling various problems, but come with advantages and disadvantages due to the challenges they all face: They must *scale up*, because legacy software systems tend to be very large, and they must be flexible and applicable in different contexts, as there is no such thing as a standard reverse engineering context, *i.e.*, every legacy system comes with its own problems and flaws. Moreover, they must be simple and straight-forward to use, because in the fast-paced software industry there is little time to reverse engineer software systems, *i.e.*, reverse engineering did not yet receive serious attention, as is also underlined by the scarce application of these approaches in the development process of software companies.

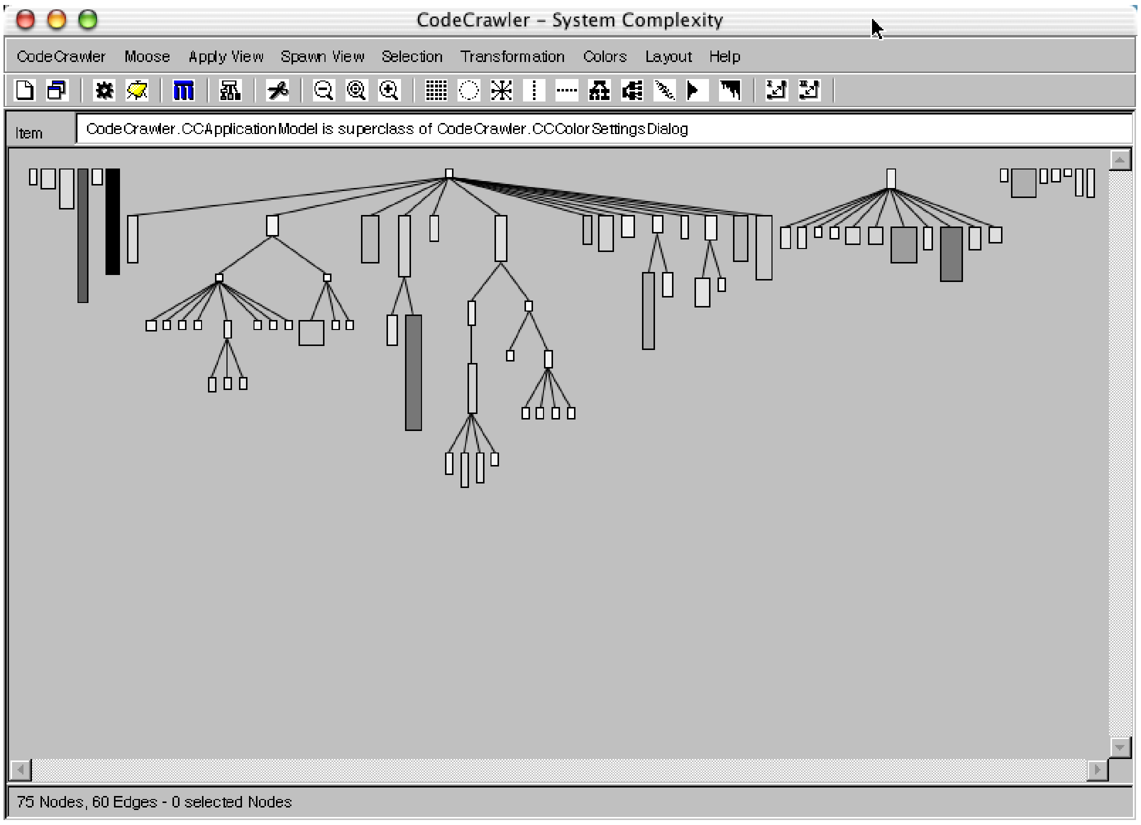
This is bound to change, as software maintenance is regarded more and more as part of the life-cycle of software systems, *e.g*, the boundary between software development and software maintenance is becoming fuzzy. This is also further emphasized by the recent emergence of lightweight, agile, and iterative development methodologies like eXtreme Programming [COCK 01, BECK 00] that support an *evolutionary* view of software: software systems are never finished, they are rather constantly under development. Lehman's software evolution laws state that software systems must be continually adapted else they become progressively less satisfactory [LEHM 85]. Put in simpler words, only dead software systems do not evolve.

1.3 Our Approach

In this thesis we propose a lightweight reverse engineering approach based on simple software visualizations enriched with software metrics, which we call *polymetric views*. Polymetric views permit to enrich a visualization with up to five different software metrics, two for the size, two for the position, and one for the color of each node. We implemented a tool called *CodeCrawler* [LANZ 03] which supports the polymetric views.

In Figure 1.1 we see CodeCrawler showing an example polymetric view called SYSTEM COMPLEXITY. It visualizes classes as nodes and inheritance relationships as edges. The class nodes are enriched with the metrics NOA (number of attributes) for the width, NOM (number of methods) for the height, and LOC (lines of code) for the color. This view permits us to answer questions about the size of the system, about the location of big classes, about the size of classes, etc.

Moreover, our visualizations are interactive, *e.g.*, the user can not only see but also interact (zoom, move, remove, hide, etc.) with the polymetric views. We believe that by making such interactions possible, the gap between the software and the reverse engineer's mental model of the software can be further narrowed. We do not think that our software visualizations alone are enough to tackle the problems of reverse engineering, but claim that our approach is aimed at supporting and complementing other techniques, like the ones listed above, in order to enhance and facilitate the comprehension of software systems. We argue that simple and lightweight reverse engineering approaches can be exploited faster than heavyweight approaches, and support existing practices instead of replacing them.


CodeCrawler – System Complexity
CodeCrawler Moose Apply View Spawn View Selection Transformation Colors Layout Help
Item

CodeCrawler.CCApplicationModel is superclass of CodeCrawler.CCColorSettingsDialog

75 Nodes, 60 Edges - 0 selected Nodes

Figure 1.1: The polymetric view SYSTEM COMPLEXITY applied on CodeCrawler.

According to the program cognition model vocabulary proposed by Littman *et al.* [LITT 96] we support an approach of understanding that is *opportunistic* in the sense that it is not based on a *systematic* line-by-line understanding but *as needed*. Moreover, to locate our approach in the general context of cognitive models [LITT 96] [VON 96], our approach is intended to support the understanding of the *implementation plans* at the language level, *i.e.*, classes and methods.

We have developed polymetric views targeting three different granularity levels at which we want to gather information about an object-oriented software system, namely at a *coarse-grained, fine-grained, and evolutionary* level.

1. **Coarse-grained polymetric views.** The target of the coarse-grained polymetric views is to visualize very large software systems, for which we seek to obtain an initial understanding of their structure and their properties. This information is useful for identifying the parts of a subject system which need to be further analyzed, and to obtain an overall view that reduces the complexity inherent in such systems. We present a selection of coarse-grained polymetric views, which transmit to the viewer a great deal of information (*e.g.*, complete systems are being visualized) in a short time span. The coarse-grained views can answer questions about the overall structure of a system, about the detection of particular software artifacts, and provide a first impression of a subject system. Furthermore based on the coarse-grained views we introduce an *approach* to guide software developers in the first steps of a reverse engineering process of an unknown system.

2. **Fine-grained polymetric views: the class blueprint.** The *class blueprint* view enables us to focus on the understanding of the core elements of object-oriented programming languages, namely classes and class hierarchies. The goal is to obtain an understanding of the inner structure of one class or several classes at once. Furthermore it it useful for detecting patterns in the implementation of

classes and class hierarchies. These patterns help to answer questions regarding the internal structure of classes and class hierarchies and are also useful for the detection of design patterns.

3. **Evolutionary software polymetric views: the evolution matrix.** The *evolution matrix* view visualizes the evolution of classes and of software systems by displaying several versions of a software system at once, thus simplifying the interpretation of the displayed information. Furthermore it is useful for detecting patterns in the evolution of classes and software systems.

In this thesis we claim that the presented polymetric views enhance the reverse engineering process and provide insights into the structure of object-oriented software software systems at various levels of understanding, namely coarse-grained understanding of large software systems, fine-grained understanding of classes and class hierarchies, and the understanding of the evolution of software systems and classes.

1.4 Contributions

The contributions of this thesis can be summarized as follows:

- The concept of a *polymetric view*, a lightweight software visualization technique enriched with software metrics information. At one glance polymetric views can transmit a great deal of information to a reverse engineer that would otherwise be hard to retrieve and/or combine.
- The presentation and discussion of several polymetric views which all target coarse-grained reverse engineering. These views help to answer questions regarding the overall structure and architecture of a system, help to detect outliers, and give a first general impression of a subject system.
- The development of a reverse engineering methodology based on the coarse-grained polymetric views which is applicable during the first stages of a reverse engineering process.
- The concept of the *class blueprint* view, a fine-grained polymetric view that visualizes the internal structure of one or more classes.
- Based on our blueprint visualization we identify and categorize several types of patterns that reveal information about implementation aspects of the classes. These patterns help to answer questions regarding the fine-grained details of an object-oriented system and provide information about classes and class hierarchies in terms of code quality, coding conventions, detection of design patterns, etc.
- The concept of the *evolution matrix* view, an evolutionary polymetric view targeted at the visualization of the evolution of object-oriented software systems.
- The evolution matrix enables us to identify patterns that reveal information about the evolution of classes and of systems.

1.5 Thesis Outline

This dissertation is structured as follows:

- In Chapter 2 we introduce the problem domains of reverse engineering and software visualization in detail and point out current approaches, as well as advantages and drawbacks of those solutions.
- In Chapter 3 we introduce and discuss the concept of the *polymetric views*. Polymetric views are lightweight software visualizations enriched with metrics information.
- In Chapter 4 we present a reverse engineering approach based on coarse-grained software visualization which mainly targets large scale software systems by means of *polymetric views*. We call the presented views *coarse-grained*, because they are mainly applicable to large software systems, *i.e.*, many of the views scale up. We present several views and discuss them in the context of a reverse engineering methodology which can be used to reverse engineer large software systems.

- In Chapter 5 we present a fine-grained approach which aims at understanding single classes and class hierarchies by means of a specific visualization called *class blueprint*. The class blueprint view supports opportunistic code-reading, *i.e.*, it helps us to understand the inner structure of one or several classes by visualizing a semantically augmented call- and access-graph of the methods and the attributes of classes. Besides the technical aspects that the class blueprint implies, we establish a vocabulary which identifies the most common and specific *visual patterns*, *i.e.*, recurrent graphical situations. This vocabulary is the basis of a language that reverse engineers can use when communicating with each other.

- In Chapter 6 we provide an approach which takes into account the evolution of software systems and is able to visualize it by means of a visualization called *evolution matrix*. The evolution matrix allows for a quick understanding of (1) the evolution of classes within object-oriented software systems and the (2) the evolution of the systems themselves. Furthermore we provide a vocabulary of class evolution behavior based on *visual patterns* we detect in the evolution matrix view.

- In Chapter 7 we conclude by summarizing the main contributions of our work and give also an outlook on possible future work in this research field.

- In Appendix A we then add some technical discussions, mainly about the implementation of our tool *CodeCrawler* and *the Moose Reengineering Environment*.

Chapter 2

Object-Oriented Reverse Engineering

2.1 Introduction

The maintenance, reengineering, and evolution of software systems has become a vital matter in today's software industry. The law of *software entropy* dictates that most systems with time tend to gradually decay in quality, unless the systems are maintained and adapted to the evolving requirements. However, many requirements (changing platforms, new functionalities demanded by the users, the fixing of errors, etc.) cannot be anticipated and are hard to fulfill because of the system's original design and architecture: systems are not prepared to support new platforms, embrace emerging standards, satisfy new customer needs, and leverage better understood technological advancements. Therefore they must be adapted to do so.

When companies face the decision to maintain and/or reengineer a software system, they must evaluate whether to rewrite the system from scratch. Sommerville [SOMM 00] and Davis [DAVI 95] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system, and it would thus seem advisable to rewrite software systems as soon as they fail to fulfill their requirements. Moreover, a lot of software systems can be upgraded, or simply be thrown away and be replaced when they no longer serve their purposes.

However, certain software systems (*legacy* software systems) are too valuable to be replaced or to be rewritten, because their sheer size and complexity makes such a feat too expensive for the owning company in terms of time and money. In such cases it is more advisable to maintain, reengineer, and evolve such systems by adapting them to new requirements [CASA 98, RUGA 98] in order to extend their lifetime and to increase the return of investment of their owners. Indeed, the longer a software system can be used, the better it pays off for the company that developed it.

The lifetime of a software system can be extended by maintaining and/or reengineering it. Both software maintenance and reengineering can be seen as part of a general software evolution process, *e.g.*, the software is evolved in order to keep its value and to cope with new requirements: Lehman's software evolution laws state software systems must be continually adapted else they become progressively less satisfactory [LEHM 85].

Before a software system can be reengineered and/or maintained the system must be *reverse engineered*, *e.g.*, a mental model of the software needs to be built which allows for taking informed decisions. Reverse engineering software systems is difficult due to their sheer size and complexity. However, it is a *prerequisite* for their maintenance, reengineering and evolution.

Maintaining and evolving existing software systems is difficult because of several reasons, such as the accelerating turnover of developers, the increasing size and complexity of software systems, and the constantly changing requirements of software systems. These *legacy systems* are large, mature, and complex software systems, which are valuable to a company and must therefore be maintained and evolved [CASA 98, RUGA 98].

However, as Parnas [PARN 94] assessed, most legacy systems suffer from typical problems:

- The original developers are no longer available, and often inexperienced developers without system or domain knowledge are assigned to maintain and evolve the systems.

- Outdated software development methods and/or programming languages have been used to originally develop the system, and it is hard to find people knowing these techniques or willing to learn them.
- Extensive modifications and patches have been applied to the system, triggering a phenomenon called *architectural drift*, which makes it hard to understand the original design of the system.
- The documentation is outdated, incomplete or completely missing.

Since legacy systems tend to be large – hundreds of thousands of lines of poorly documented code are no exception – there is a definite need for effective approaches which help in program understanding and problem detection. We have focused ourselves on object-oriented legacy systems [WILD 92], mainly because most current systems are written using this paradigm, and because it is not *age* that turns a piece of software into a legacy system, but the *rate* at which it has been developed and adapted [DEME 02].

Moreover, compared with procedural systems, the reengineering and reverse engineering of *object-oriented* software systems poses many additional challenges [WILD 92], such as polymorphism, late-binding, incremental class definitions by means of inheritance, the dynamic semantics of *self* and *this*. Furthermore, in object-oriented systems the domain model is spread over classes residing in different hierarchies and/or subsystems. Indeed, the problem of reverse engineering object-oriented software systems needs to be tackled.

Structure of the chapter. In Section 2.2 we discuss reverse engineering as part of the reengineering life-cycle of software, present a state-of-the-art and discuss the problems and challenges that reverse engineering poses, as well as the constraints it has. We then introduce our approach based on a combination of software visualization and software metrics in Section 2.3.

2.2 Object-Oriented Reverse Engineering

Reverse engineering is part of the *reengineering life-cycle* [RAJL 00] [DEME 02]. Chikofsky and Cross define reengineering as “the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.” [CHIK 90]. *Forward* engineering, on the other hand, is defined as “the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system” [CHIK 90].

Therefore, if forward engineering is about moving from high-level views of requirements and models toward concrete realizations, then reverse engineering is about going backward from some concrete realization to more abstract models, and reengineering is about transforming concrete implementations to other concrete implementations.

Figure 2.1 illustrates this idea:

- *Forward engineering* can be understood as being a process that moves from high-level and abstract models and artifacts to increasingly concrete ones.
- *Reverse engineering* reconstructs higher-level models and artifacts from code.
- *Reengineering* is a process that transforms one low-level representation to another, while recreating the higher-level artifacts along the way [DEME 02].

2.2.1 General Approaches to Reverse Engineering

The goal of a person that is reverse engineering a software system is to build progressively refined mental models of the system [STOR 99] to be able to make informed decisions regarding the software. While this is not a complex problem for small software systems, where code reading and inspection is often enough, in the case of legacy software systems which tend to be large – hundreds of thousands or millions of lines of poorly documented code are no exception – this becomes a hard problem because of their sheer

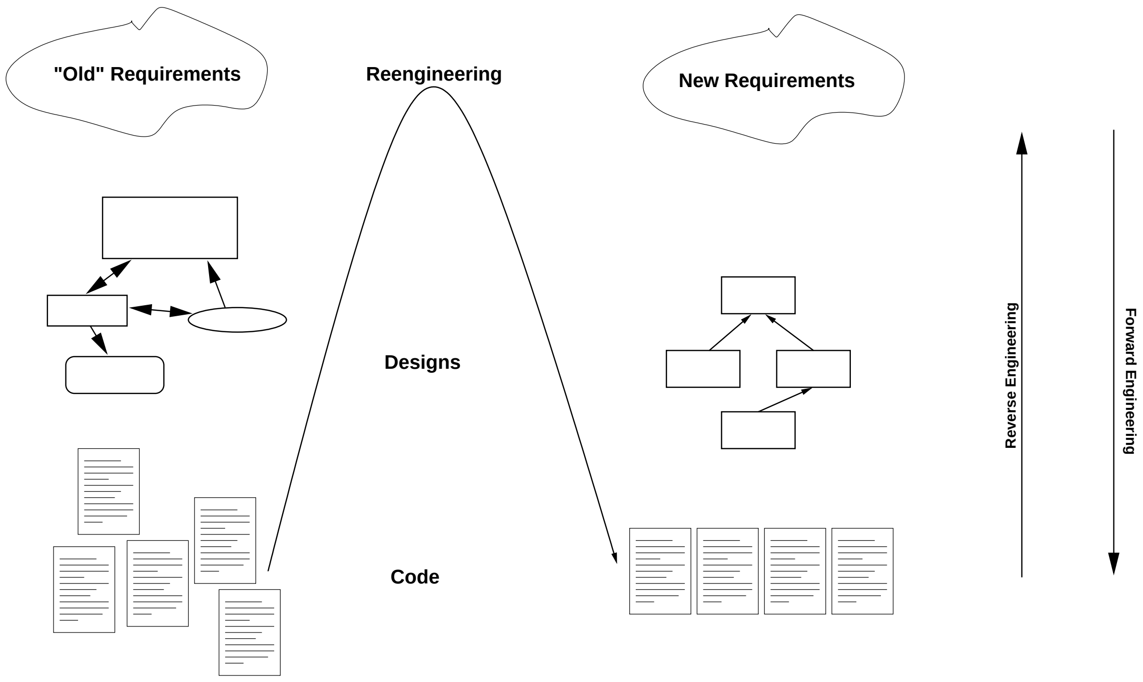


Figure 2.1: The reengineering life-cycle of software.

size and complexity, and because of the problems afflicting such systems [PARN 94]. In order to build a progressively refined mental model of a software system, the reverse engineer must gather information about the system which helps him in this process.

There are many approaches to reverse engineering software systems, such as:

- reading the existing documentation and source code. Various people have investigated code inspection, code reading, and code review practices [DEKE 02, HEND 02, DEME 02]. Using this approach is difficult when the documentation is obsolete, incorrect or not present at all. Reading the source code is a widely used practice, but does not scale up, as reading millions of lines of code would take weeks or months without necessarily increasing the understanding of the system by the reader. Moreover, at the beginning of a reverse engineering process one does not seek detailed information, but rather wants to have a general view of the system.

- running the software and/or generate and analyze execution traces. The use of dynamic information, *e.g.*, information gathered during the execution of a piece of software, has also been used in the context of reverse engineering [RICH 99, JERD 97, RICH 02], but has drawbacks in terms of scalability (traces of a few seconds can become very big) and interpretation (thousands of message invocation can hide the important information one is looking for).

- interviewing the users and developers. This can give important insights into a software system, but is problematic because of the subjective viewpoints of the interviewed people and because it is hard to formalize and reuse these insights. Moreover, it can be hard to find developers that have been part of the development team over long periods of time and thus possess knowledge about a software system's complete lifetime.

- using various tools (visualizers, slicers, query engines, etc.) and techniques (visualization, clustering, concept analysis, etc.) to generate high-level views of the source code. Tool support is provided by the research community in various ways, and visualization tools like Rigi [MÜ 86] and ShrimpViews [STOR 95] are widely used.

- analyzing the version history, as for example done by Jazayeri [JAZA 99]. Still a young research field, understanding the evolution of a piece of software is done using techniques like graph rewriting, visualization, concept analysis, clustering, and data mining. The insights gained are useful to understand the past of a piece of software and to possibly predict its evolution in the future.
- assessing a software system and its quality by using software metrics. Software metrics tools are used to assess the quality and quantity of source code by computing various metrics which can be used to detect outliers and other parts of interest, for example cohesive classes, coupled subsystems, etc.

In this thesis we present an approach which using software visualization enriched with metrics information enables us to generate high-level views and thus build a mental model of a system.

Although the term 'legacy system' is often associated with systems written in older programming languages, recent object-oriented systems suffer from similar problems. Moreover, reengineering is becoming part of modern software development processes and thus also plays a major role in the development of systems written in recent object-oriented programming languages like Java.

2.2.2 Challenges and Goals in Object-Oriented Reverse Engineering

Challenges

Compared with procedural systems, the reverse engineering of object-oriented software systems poses many additional challenges [WILD 92]. We list some of them:

- Polymorphism and late-binding make traditional tool analyzers like program slicers inadequate. Data-flow analyzers are more complex to build especially in presence of dynamically typed languages.
- The use of inheritance and incremental class definitions, together with the dynamic semantics of *self* and *this*, make applications more difficult to understand.
- The domain model of the applications is spread over classes residing in different hierarchies and/or subsystems and it is difficult to pinpoint the location of a certain functionality.
- Contrary to procedural systems, where a top-down reverse engineering approach can work because of the structured decomposition of an application, in the case of object-oriented systems the first question a reverse engineer has to answer is where to start the reverse engineering process.

Apart from these problems, the increased power and flexibility that object-oriented programming brought to developers can also be harmful, as in the case of misuse of inheritance or the violation of encapsulation. This is mainly due to the fact that programmers still have problems to completely understand the concepts behind object-oriented programming. Casais [CASA 98] states that "[...] experience demonstrates that software developers have trouble imparting object-oriented applications or components with the generality and adaptability needed for diverse and changing requirements."

Goals

Chikofsky and Cross state that *"The primary purpose of* ***reverse engineering*** *a software system is to increase the overall comprehensibility of the system for both maintenance and new development"* [CHIK 90]. They list six key reverse engineering objectives:

1. Cope with complexity
2. Generate alternate views
3. Recover lost information
4. Detect side effects

5. Synthesize higher abstractions
6. Facilitate reuse

Before starting a reverse engineering process it is therefore essential to decide which primary goals to pursue and which ones are only of secondary importance. Moreover, the listed objectives must be broken down into concrete research questions whose answer we are seeking. The approaches listed previously are useful to follow one or more of these key objectives, for example visualization can generate alternate views, while software metrics can recover lost information. We argue that these objectives are too general to be realistic goals, but are composed of smaller and much more concrete objectives.

2.3 Our Work: Scope, Constraints, and Goals

As we have seen, the goal of reverse engineering a large legacy software system is to build a progressively refined mental model of the system in order to make informed decisions regarding the system. Building such a mental model requires gathering information about the system. We have seen that there are various approaches to gather the needed information. We argue that the necessary information resides at various granularity levels. In the case of object-oriented systems we think there are three complementary granularity levels, namely coarse-grained information about large software systems, fine-grained information of classes and class hierarchies, and the information about the evolution of software systems and classes.

In this thesis we propose a lightweight approach that permits to retrieve the information at these various levels. We think the approach must be lightweight at this time, as industry is resistant against disrupting their development process with new approaches. A lightweight approach, which does not replace existing approaches, but complements them, has a greater chance of success.

2.3.1 Scope

Our work combines the two techniques of software visualization and software metrics and uses this combination to enable us to reverse engineer object-oriented software systems.

Software Visualization

Software visualization is defined as “the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software.” [Sta 98]. It is a specialization of *information visualization*, whose goal is to visualize any kind of abstract data, while in software visualization the sole focus lies on visualizing software.

Information Visualization. Information visualization is defined as “the use of computer-supported, interactive, visual representations of abstract data to amplify cognition.” [Car 99]. It derives from several communities. Starting with Playfair (1786), the classical methods of plotting data were developed. In 1967, Jacques Bertin, a French cartographer, published his theory in *the semiology of graphics* [BERT 74]. This theory identifies the basic elements of diagrams and describes a framework for their design. Edward Tufte published a theory of data graphics that emphasized maximizing the density of useful information [TUFT 90, TUFT 97]. Both Bertin’s and Tufte’s theories have been influential in the various communities that led to the development of information visualization.

The goal of information visualization is to *visualize any kind of data*. Note that the above definition of information visualization does not necessarily imply the use of vision for perception: visualizing does not necessarily involve *visual* approaches, but any kind of *perceptive* approach. Data can be perceived by a person by using the senses at our disposition, *i.e.*, apart from seeing the data, a person can also hear it (information auralization) and/or touch it (by using virtual reality technology). It must be emphasized that most information visualization systems involve using computer graphics which render the data using 2D- and/or 3D-views of the data. Applications in information visualization are so frequent and common, that most people do not notice them: examples include metereology (weather maps), geography (street maps), geology, medicine (computer-aided displays to show the inner of the human body), transportation (train tables and metro maps), etc.

In short, information visualization is about visualizing almost any kind of data in almost any kind of way, while software visualization is about visualizing software.

The field of software visualization can be divided in two separate areas [Sta 98]:

1. *Program visualization* is the visualization of actual program code or data structures in either static or dynamic form. The approach of the polymetric views presented in this thesis belongs to a sub-area of program visualization, namely *static code visualization*, because we visualize (object-oriented) source code by using only information which can be *statically* extracted from the source code without the need to actually run the system.

2. *Algorithm visualization* is the visualization of higher-level abstractions which describe software. A good example is *algorithm animation*, which is the dynamic visualization of an algorithm and its execution. This was mainly done to explain the inner working of algorithms like sort-algorithms. In the meantime this discipline has lost importance, mainly because the advancement in computer hardware and the possibility to use standard libraries containing such algorithms have shifted the focus away from the implementation of such algorithms.

Software visualization and reverse engineering. Software visualization has been widely used by the reverse engineering research community during the past two decades [Sta 98, STOR 97, STOR 98, M.-A 01]. Many of them provide ways to uncover and navigate information about software systems. We omit an in-depth discussion about software visualization work here, as we discuss it in the subsequent chapters in a more context-relevant way. Our approach does not differ very much from existing software visualization techniques, but adds a dimension of understanding more by allowing us to enrich the visualization with software metrics information.

Software Metrics

"What is not measurable make measurable." (Galileo Galilei)

Metrics have long been studied as a way to assess the quality and complexity of software [FENT 96], and recently this has been applied to object-oriented software as well [LORE 94] [HEND 96].

In a reverse engineering context software metrics are interesting because they can be used to assess the quality and complexity of a system and because they are known to scale up. Furthermore, metrics are a good means to control the quality and the state of a software system during the development process [FENT 96]. However, metric measurements often come in huge tables that are hard to interpret, and this is even more difficult when metrics are combined to generate yet other metrics.

Formally, metrics measure certain properties of a software system 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 system 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 both be planned and performed better.

Metrics can be divided in two groups [LORE 94]:

1. *Design Metrics.* These metrics are used to assess the size and in some cases the quality, size and complexity of software. They take a look at the quality of the project's design at a particular point in the development cycle. Design metrics tend to be more locally focused and more specific, thereby allowing them to be used effectively to directly examine and improve the quality of the product components.

2. *Project Metrics.* They deal with the dynamics of a project, with what it takes to get to a certain point in the development life cycle and how to know you're there. They can be used in a predictive manner, for example to estimate staffing requirements. Being at a higher level of abstraction, they are less prescriptive and more fuzzy but are more important from an overall project perspective.

In this work we have decided to use *direct measurement* metrics, *e.g.*, metrics which can be computed directly from the source code without using any other kind of information. We have taken choice in order to comply with the principle to use a lightweight approach.

2.3.2 Constraints

We developed our approach in the context of the European Esprit project FAMOOS, whose main results have been summarized in a reengineering handbook [Duc 99] which was the basis for a book on object-oriented reengineering patterns [DEME 02]. The goal of the project was to reengineer several large industrial object-oriented software systems. The industrial setting of the FAMOOS project introduced the following constraints:

- **Simplicity.** In software industry, reengineers face many problems, *i.e.*, short time constraints, little tool support, and limited manpower. It is for this reason that we wanted our results to be reproduceable by software engineers at their workplace, without having to rely on complex or expensive tools. Moreover, by choosing a simple, lightweight approach, we were able to get results quickly, in order to evaluate whether certain ideas were viable or not.
- **Scalability.** We wanted to make sure that our approach could handle the size of industrial systems, which can be of several millions of lines of code. Scalability is on one hand guaranteed by the use of software metrics, since metrics can be computed independently from the size of the system. On the other hand our approach allowed us to generate, test and accept/reject new ideas in short iteration cycles. After starting the development of our approach and the implementation of our tools to validate the approach, we repeatedly tested them in industrial settings to see whether they were actually viable and could indeed scale up.
- **Language Independence.** In order to handle software systems written in different languages, we developed FAMIX [DEME 01, TICH 01], a language independent metamodel. Our implementation in Smalltalk of the FAMIX metamodel, called the Moose Reengineering Environment [DUCA 00], is discussed in Appendix A.
- **Short Time.** The amount of time to reverse engineer the subject systems was very short, and ranged from one or two days to maximum one week. This is too short for a complete reverse engineering, and we thus aimed more at getting a *raw understanding* of the systems.

2.3.3 Goals

When reverse engineering a software system it is of key importance to have clear goals in mind. The six objectives listed by Chikofsky and Cross [CHIK 90] are too general to be directly usable as concrete goals. We make here a canonical list of goals we want to achieve when reverse engineering a system, *e.g.*, the following is a list of questions and goals we want to test our approach against. Moreover, we can divide the goals in three complementary sets which target different levels of understanding, namely coarse-grained, fine-grained, and evolutionary understanding.

In the context of object-oriented legacy systems we settled on the following goals:

- Coarse-grained Goals:
  1. Assess the overall quality of the system.
  2. Gain an overview of the system in terms of size, complexity, and structure.
  3. Locate and understand the most important classes and inheritance hierarchies, *i.e.*, find the classes and hierarchies that represent a core part of the system's domain and understand their design, structure in terms of implementation, and purpose in terms of functionality.
  4. Identify exceptional classes in terms of size and/or complexity compared to all classes in the subject system. These may be candidates for a further inspection or for the application of refactorings.

5. Identify exceptional methods in terms of size and/or complexity compared to the average of the methods in the subject system. These may be candidates for a further inspection regarding duplicated code or for the application of refactorings.
6. Locate unused, *e.g.*, dead code. This can be unused attributes, methods that are never invoked or that have commented method bodies, unreferenced classes, etc.

- Fine-grained Goals:

  1. Understand the concrete implementation of classes and class hierarchies, and detect common patterns or coding styles. Look for signs of inconsistencies like the use of accessors.
  2. Identify the possible presence of design patterns or occasions where design patterns could be introduced to ameliorate the system's structure.
  3. Build a mental image of a class in terms of method invocations and state access.
  4. Understand the class/subclass roles.
  5. Identify key methods in a class.

- Evolutionary Goals:

  1. Understand the evolution of object-oriented software systems in terms of size and growth rate.
  2. Understand at which point in time classes have been introduced into a system and at which moment they have been removed.
  3. Understand how classes grow and shrink in terms of the number of methods and the number of attributes.
  4. See if there are patterns in the evolution of classes. Such patterns help to understand the condition of a class in a time perspective, *e.g.*, how resistant to software evolution processes is a class, is it changed with every release of a system, or are there classes which are virtually immune to software evolution?

The result of a reverse engineering process is therefore not necessarily a list of problematic classes or subsystems, even if the identification of possible design defects is a valuable piece of information. The goal of a reverse engineer is to understand the overall structure of the application, to gain a better understanding of the inheritance relationships between classes, and to gain an overview of the methods and the way they are organized. Indeed, we are looking for the bad use as well as the good use of object-oriented design. In that sense, knowing that an inheritance hierarchy is well designed is also valuable information.

Moreover, in a reengineering context the fact that a class may have a design problem does not necessarily imply that the class should be redesigned. Indeed, if a badly designed class or subsystem accomplishes the work it has been assigned to, without having a negative impact on the overall working of the system, there is no point in changing it. However, being aware of such information is still valuable for getting a better mental model of the system.

2.4 Conclusions

In this chapter we analyzed the problems of reverse engineering in general and of reverse engineering for object-oriented systems in particular. We have seen that the maintenance and evolution of legacy software systems constitute major problems that software engineers need to tackle, because the systems in question are too complex and too valuable to be replaced or to be rewritten from scratch. We discussed the limits and benefits of present approaches to solve those problems and then positioned our own approach (presented in detail in Chapter 3), which is a lightweight combination of software visualization and software metrics.

From a common set of abstract reverse engineering objectives we derived a set of goals that we want to obtain and against which we will evaluate our work.

Chapter 3

The Polymetric View

In this chapter we present the different aspects of our approach, which ultimately leads to the construction of lightweight software visualizations enriched with software metrics. We call these visualizations *polymetric views*.

Structure of the chapter. First we present how we use metrics to enrich our visualizations (Section 3.1). Then we present a selection of the metrics at our disposal (Section 3.2). Finally we present which ingredients are needed to yield visualizations, illustrate two examples of such polymetric views, and discuss how a polymetric view is to be interpreted.

3.1 The Principle

We use two-dimensional visualizations to display software. More precisely we use nodes (rectangles) to display software entities or abstractions of them, while we use edges to represent relationships between the entities. This is a widely used practice in information visualization and software visualization tools. Ware claims that "other possible graphical notations for showing connectivity would be far less effective" [WARE 00]. We enrich this basic visualization technique by rendering up to 5 metric measurements on a single node simultaneously, as we see in Figure 3.1.

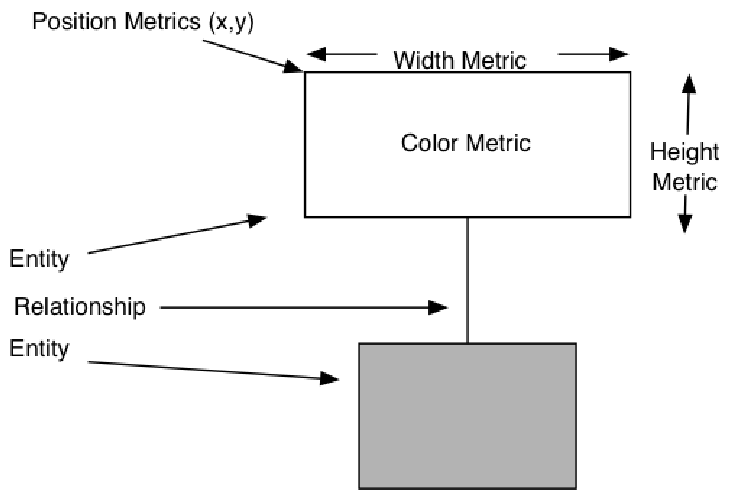


Figure 3.1: Up to 5 metrics can be visualized on one node.

- **Node Size.** The width and the height of a node can each render one metric measurement. The bigger these measurements are, the bigger the node is in one or both of the dimensions.
- **Node Color.** The color interval between white and black can be used render another metric measurement. The convention is that the higher the metric value is, the darker the node is. Thus light gray represents a smaller metric measurement than dark gray. We opted against using different colors, because nominal colors cannot reflect quantities. Tufte [TUFT 01] states that "Despite our experiences with the spectrum in science textbooks and rainbows, the mind's eye does not readily give a visual ordering to colors. Because they do have a natural visual hierarchy, varying shades of gray show varying quantities better than color". There are exceptions to this rule, for example weather maps use a spectrum which ranges from blue to red to denote warm and cold temperatures, although in this case the distinction between warm and cold introduces again a nominal ordering.
- **Node Position.** The X and Y coordinates of the position of the node can also reflect two metrics measurements. This requires the presence of an absolute origin within a fixed coordinate system. Not all layouts can exploit position metrics, as some of them implicitly dictate the position of the nodes (*e.g.*, a tree layout).

In measurement theory, this procedure of rendering metrics on two-dimensional nodes is called *measurement mapping*, and fulfills the representation condition, which asserts that "a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations" [FENT 96]. In other words, if a number a is bigger than a number b, the graphical representation of a and b must preserve this fact.

Because we want to have interactive visualizations, *e.g.*, the viewer must be able to interact with the visualization, we defined our measurement mapping function considering a real-world issue: in order for the user to be able to click on a node, the node must have a certain size. On the other hand the size of the node must render the underlying metric measurement as truthfully as possible. The first idea which comes to mind is a direct one-to-one mapping. However, this idea must be rejected because if a measurement is zero the node will have no dimension. We have considered several possible solutions [LANZ 99] for this problem, and have finally settled on defining a minimal node size (MNS) value, to which the metric measurement is directly added. We settled on having MNS = 4, this value can however be easily changed. In Table 3.1 we see how our measurement mapping function behaves on certain input values by assuming that MNS = 4.

Metric Measurement	**Resulting Node Width/Height**
0	4
1	5
2	6
5	9
10	14
100	104

Table 3.1: An exemplification of our measurement mapping function based on a minimal node size (MNS) of 4.

3.2 Software Metrics

In our polymetric views we make extensive use of object-oriented software metrics. In the wide array of possible metrics [LORE 94] [HEND 96] [FENT 96] we selected *design metrics*, *i.e.*, metrics that can be extracted from the source code entities themselves. These metrics are usually used to assess the size and in some cases the quality and complexity of software.

The metrics we use are termed *direct measurement* metrics because their computation involves no other attributes or entities [FENT 96]. We don't make use of *indirect measurement* where metrics are combined to generate new ones, because the *measurement mapping* presented in the previous section works best with direct measurements. Examples of indirect metrics include *programmer productivity*, *defect detection density* or *module defect density*, as well as more code-oriented ones like *CBO* and *RFC*, presented in [CHID 94]. We chose to use metrics that can be extracted from source code entities, and which have a simple and clear definition. As such we don't use *composite metrics*, which raise the issue of dimensional consistency [HEND 96].

Class Metrics	
Name	**Description**
HNL	Number of classes in superclass chain of class
NAM	Number of abstract methods
NCV	Number of (static) class variables
NIA	Number of inherited attributes
NIV	Number of instance variables
NME	Number of methods extended, *i.e.*, redefined in subclass by invoking the same method on a superclass
NMI	Number of methods inherited, *i.e.*, defined in superclass and inherited unmodified by subclass
NMO	Number of methods overridden, *i.e.*, redefined compared to superclass
NOA	Number of attributes (NOA = NIV + NCV)
NOC	Number of immediate subclasses of a class
NOM	Number of methods
WLOC	Sum of LOC over all methods
WMSG	Sum of message sends in a class
WNMAA	Number of all accesses on attributes
WNOC	Number of all descendant classes
WNOS	Sum of statements in all method bodies of class
WNI	Number of invocations of all methods

Table 3.2: A list of the class metrics used in this thesis.

Method Metrics	
Name	**Description**
LOC	Method lines of code
NMA	Number of methods added, *i.e.*, defined in subclass and not in superclass
MHNL	Class HNL in which method is implemented
MSG	Number of method message sends
NOP	Number of (input) parameters
NI	Number of invocations of other methods within method body
NMAA	Number of accesses on attributes
NOS	Number of statements in method body

Table 3.3: A list of the method metrics used in this thesis.

In Table 3.2, Table 3.3, and Table 3.4 we list all the software metrics mentioned in this thesis. The metrics are divided into three groups, namely class, method and attribute metrics, *i.e.*, these are the entities the metric measurements are assigned to. Since one of our main constraints is to reengineer systems written in different object-oriented languages we have chosen to include in our metrics engine metrics

Attribute Metrics	
Name	**Description**
AHNL	Class HNL in which attribute is defined
NAA	Number of times directly accessed. Note that NAA = NGA + NLA
NGA	Number of direct accesses from outside of its class
NLA	Number of direct accesses from within its class

Table 3.4: A list of the attribute metrics used in this thesis.

whose computation does not depend on any language-specific features, but can be based directly on our language-independent metamodel, which we present in Section A.

3.3 The Actual Visualization: A Polymetric View

An actual polymetric view depends on three ingredients, (1) *a layout*, (2) *a set of metrics*, and (3) *a set of entities*.

1. **A layout.** A layout takes into account the choice of the displayed entities and their relationships and issues like whether the complete display should fit on the screen, whether space should be minimized, whether nodes should be sorted, etc. Some layouts make sense for all purposes, while others are better suited for special cases (*e.g.*, a tree layout is better suited for the display of an inheritance hierarchy than a circle layout).

   We implemented a small set of simple layouts [LANZ 99] and list below the essential ones. As part of our lightweight approach, we chose to implement only simple ones, although more advanced and powerful layouting techniques [BATT 99] are also interesting in this context. The essential layouts we used are Tree, Scatterplot, Histogram, Checker and Stapled. We have chosen to use these layouts because of their simplicity and because of the advantages discussed below.

   - *Tree.* Positions all entities according to some hierarchical relationship. See Figure 3.2 for an example. This layout is essential to visualize hierarchical structures. In the case of object-oriented programming languages this applies especially for classes and their inheritance relationships.
   - *Scatterplot.* Positions nodes in an orthogonal grid (origin in the upper left corner) according to two measurements. Entities with two identical measurements will overlap. This algorithm is useful for comparing two metrics in large populations. See Figure 4.10 for an example. This layout is very scalable, because the space it consumes is due to the measurements of the nodes and not to the actual number of nodes.
   - *Histogram.* Positions nodes along a vertical axis depending on one measurement. Nodes with the same measurement are then positioned in rows, one beside the other. See Figure 4.6 for an example. This layout is useful for analyzing the distribution of a population with regard to a certain metric.
   - *Checker.* Sorts nodes according to a given metric and then places them into several rows in a checkerboard pattern. It is useful for getting a first impression, especially for the relative proportions between the measurements of the visualized nodes. See Figure 4.3 for an example. This layout's advantage is that is uses little space to layout large numbers of nodes. Moreover, since the nodes are sorted according to a certain metric, it can also be used to easily detect outliers.
   - *Stapled.* Sorts nodes according to the width metric, renders a second metric as the height of a node and then positions nodes one besides the other in a long row. This layout is used to detect exceptional cases for metrics that usually correlate, because it normally results in a steady declining staircase, while exceptions break the steady declination. See Figure 4.9 for an example.

2. **The metrics.** We incorporate up to 5 metrics selected from the set of metrics presented in Section 3.2 into a polymetric view, as we have seen in Section 3.1. The choice of the metrics heavily influences the resulting visualization, as well as its interpretation.

3. **The entities.** Certain views are better suited for small parts of the system, while others can handle a complete large system. The reverse engineer must choose which parts or entities of the subject system he wants to visualize. These choices are part of the methodology discussed in depth in Chapter 4.

3.4 Two Example Polymetric Views

The System Complexity View

Figure 3.2 provides a first example: it shows a tree layout of nodes enriched with metrics information. The nodes represent classes, while the edges represent the inheritance relationships between them. The size of the nodes reflects the number of attributes (width) and the number of methods (height) of the classes, while the color tone represents the number of lines of code of the classes. In this case the position of the nodes does not reflect metric measurements, as the nodes' position is implicitly given by the tree layout.

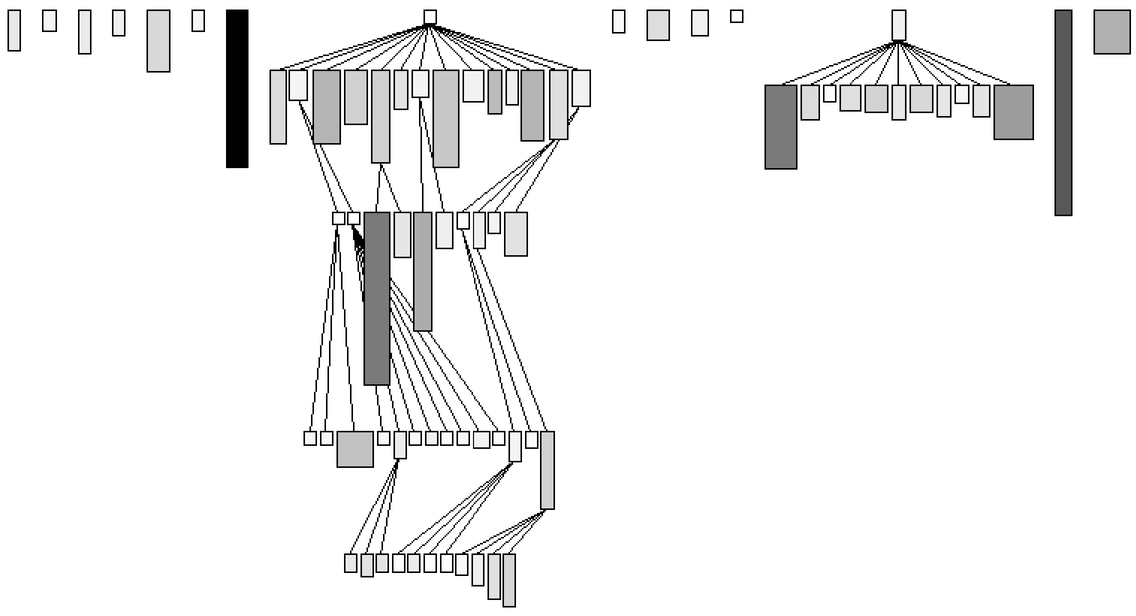


Figure 3.2: The SYSTEM COMPLEXITY view. This visualization of classes uses a tree layout. The edges represent inheritance relationships. The metrics we use to enrich the view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. The color shade represents WLOC (the number of lines of code of a class).

The combination of the tree layout, the metrics mentioned above and the selection of classes as nodes and inheritance relationships as edges yields a polymetric view that we call SYSTEM COMPLEXITY view. It visualizes classes as nodes, while the edges represent inheritance relationships. The metrics we use to enrich this view are NOA (the number of attributes of a class) for the width and NOM (the number of methods of a class) for the height. The color shade represents WLOC (the number of lines of code of a class). We discuss this view's properties and its interpretation in a reverse engineering context in more detail in Chapter 4.

The Method Efficiency Correlation View

Figure 3.3 provides a second example: it shows a correlation layout of methods nodes. In this case we do not use any size or color metrics, but only the two position metrics: the horizontal position of the nodes represents LOC (the lines of code), while the vertical position of the nodes represents NOS (the number of statements in the method body).

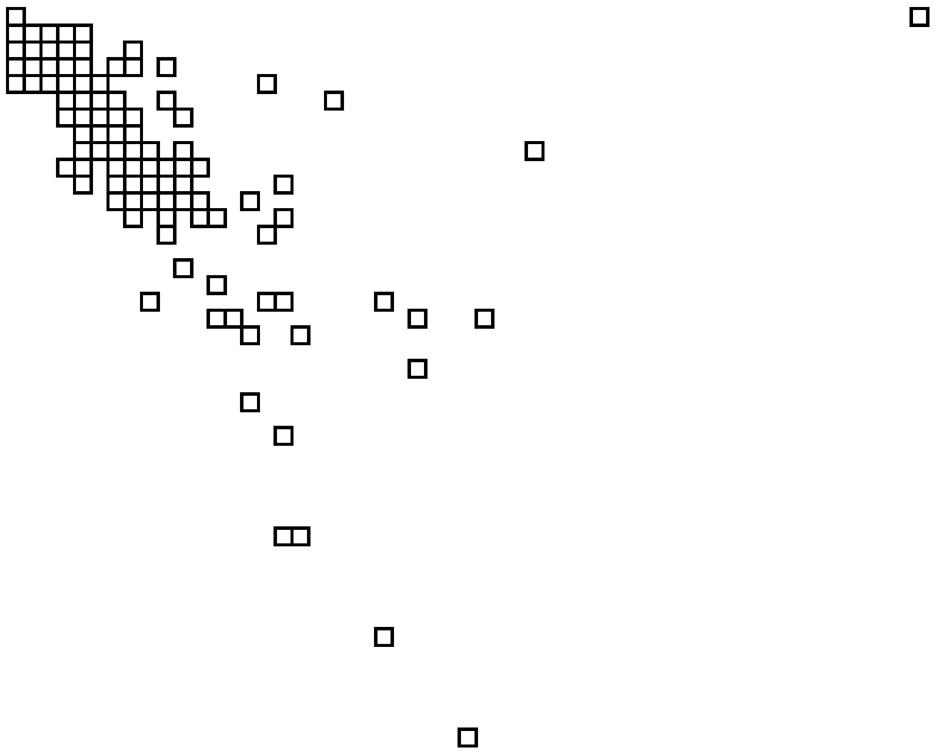


Figure 3.3: The METHOD EFFICIENCY CORRELATION view. This visualization of methods uses a correlation layout. The metrics we use to enrich the view are LOC (the number of line of code of a method) for the horizontal position and NOS (the number of statements in a method body) for the vertical position.

The combination of the correlation layout, the used position metrics and the choice of visualizing methods as nodes yields a view which we call METHOD EFFICIENCY CORRELATION view. We discuss this view's properties and its interpretation in a reverse engineering context in more detail in Chapter 4.

3.5 Interpretation of a Polymetric View

The polymetric views are revealers of *symptoms* which reside at a purely visual level, *i.e.*, they can be small dark nodes or large nodes, or even nodes at a certain position. These symptoms provide information about the subject system and support the decision process of which next view should be applied on which part of the system by the reverse engineer. Not all views lead to other views, but they may also result in specific reengineering actions that represent the next logical step after the detection of defects. For example detecting a "god class", defined by Riel [RIEL 96] as a class that has grown over the years ending up with too many responsibilities, may lead to a necessary splitting of the class. For example, long methods can be analyzed to see if they contain duplicated code or if they can be split up into smaller, more reusable methods [ROBE 97], etc.

3.6 Useful Polymetric Views

Since our approach allows one to combine a subset of the presented metrics with a layout algorithm on any kind of software artifacts, there is a great number of possible views. However, many of those are similar to others (*e.g.*, by exchanging the width and height metrics) and many others do not help the reverse engineering process. We identified a number of *useful* views, *i.e.*, polymetric views that are useful for the reverse engineering process, and present a subset of them in the next chapter.

3.7 Discussion

We shortly discuss several questions that may arise regarding the polymetric views.

- **2D vs 3D.** One may argue that using three-dimensional displays instead of two-dimensional ones could further increase the amount of information conveyed by the polymetric views. Brown and Najork [Sta 98] advocate the use of 3D over 2D in software visualizations for the following reasons:

  1. *Expressing fundamental information about structures that are inherently two-dimensional.* This means we could visualize yet another metric. However, our experiments and experience have shown that the meaningful combination of metrics and visualization techniques already put a strain on the viewer, the gain that would result from having the possibility to visualize a metric more would thus only be marginal.
  2. *Uniting multiple views of an object.* This means that 3D can be used as a multiple 2D view. In our visualizations we use multiple windows to reach the same effect.
  3. *Capturing a history of a two-dimensional view.* In Chapter 6 we present and discuss a polymetric view which allows us to visualize the evolution of software entities without needing to use the third dimension.

  The main reason why we chose not to use three-dimensional displays is that we consistently wanted to use lightweight approaches, and the added complexity in terms of graphical output and navigation would have contradicted this.

- **Composite metrics.** One could argue that the more complex and expressive the metrics are, the more they information they convey. For example we could use metrics like LCOM (low cohesion of methods) [CHID 94] to display classes.

  However, although this may yield interesting results (as we discuss in the chapter on future work), we believe that using direct measurement metrics which can be extracted from the source code without needing other information complies more with our lightweight approach and also prevents us from meddling with composite metrics, many of which still lack a clear and widely recognized definition. Moreover, composite metrics are often not natural numbers, but only fractions (for example between 0 and 1), and such numbers would need to be multiplied in order to have a visual impact on our views. However, this would not fit our measurement mapping approach described previously.

- **Other Visual Variables**. Bertin [BERT 74] established a vocabulary of visual variables which can be used to encode information. In the polymetric views we use for example the variables of size, color, and position. We could enhance the polymetric views even further by using other visual variables like shape (class nodes look differently than method nodes, for example with rounded corners), texture design, texture orientation, etc.

  This is part of our future work, Bertin and Tufte have both shown that one can find a better-than-average way of displaying information. Moreover, the number of polymetric views that we found by using 5 visual variables is already great, and adding more visual variables must be done carefully to prevent a visual overload.

Chapter 4

Coarse-grained Software Visualization

4.1 Introduction

Reverse engineering large object-oriented software systems is a task which needs to be performed systematically, otherwise one risks getting lost in unimportant details. Since the systems in question are too large to be understood by reading the code, there is a need for a systematic approach. Moreover, since the object-oriented paradigm does not support a reading order (*i.e.*, the domain model is distributed across classes, hierarchies, and subsystems without an explicit beginning or end), as in the case of procedural programming languages, the reverse engineer needs to know where to start looking into the system in order to understand its structure. The goal of the first phase of a reverse engineering (*e.g.*, the first week) is to gain a general impression of the system (what are we looking at?) and to get to know the particularities of the system (what are we looking for?). This goal can be broken down into concrete questions whose answer we are seeking at the beginning, which is a subset of the goals identified in Chapter 2.

Summary. In this chapter we try to answer a set of concrete reverse engineering questions by using coarse-grained polymetric views. We present several polymetric views [1], grouped in clusters, which are useful for the reverse engineering of large object-oriented software systems. With these coarse-grained polymetric views we are targeting the first phase (*e.g.*, the first week) of a reverse engineering process, because in this phase a reverse engineer forms his first mental picture of the system [STOR 99]. We call the views presented here *coarse-grained*, because they are mainly applicable to complete and large software systems, *i.e.*, many of these views scale up. The grouping of the views into clusters eases the reading, but does also provide guidelines of the context in which certain polymetric views can be applied. Moreover, we present a reverse engineering approach based on the polymetric views, which not only discusses what to look for in certain views, but also suggests which polymetric views to apply in which situations.

Contributions. The contributions of this chapter are the following:

- The definition and description of several coarse-grained *polymetric views*, lightweight software visualizations enriched with software metrics information. By applying these views to an object-oriented software system, we show that we can answer a set of concrete reverse engineering questions.
- The presentation of a reverse engineering approach based on these polymetric views, which targets the first phase of a reverse engineering process, and which is useful to guide a reverse engineer.

Structure of the chapter. In Section 4.2 we start by discussing the goals of the process that must take place in order to reverse engineer a large software system. We then motivate the need for a reverse engineering approach and introduce our own approach. In Section 4.3 we exemplify it by applying several polymetric views on the same case study and by discussing the views in detail. We then make an evaluation

[1]This chapter is an extended version of the article *Polymetric Views - A Lightweight Visual Approach to Reverse Engineering*, accepted for publication in the journal IEEE Transactions on Software Engineering, IEEE Press.

of the case study and report on our industrial experiences. Finally we discuss related work (Section 4.4) and then conclude in Section 4.5 by discussing our findings and evaluating future work.

4.2 The Reverse Engineering Process

Before starting a reverse engineering, it is essential to decide which primary goals to pursue and which ones are only of secondary importance. In the following, we present a short canonical list of primary goals we have inferred from our own reverse engineering experience, and which is a subset of the goals listed in Chapter 2. From the first phase of a reverse engineering process we expect to obtain the following results:

- Assess the overall quality of the system
- Gain an overview of the system in terms of size, complexity, and structure.
- Locate and understand the most important classes and inheritance hierarchies, *i.e.*, find the classes and hierarchies that represent a core part of the system's domain and understand their design, structure in terms of implementation, and purpose in terms of functionality.
- Identify exceptional classes in terms of size and/or complexity compared to all classes in the subject system. These may be candidates for a further inspection or for the application of refactorings.
- Identify exceptional methods in terms of size and/or complexity compared to the average of the methods in the subject system. These may be candidates for a further inspection regarding duplicated code or for the application of refactorings.
- Locate unused, *e.g.*, dead code. This can be unused attributes, methods that are never invoked or that have commented method bodies, unreferenced classes, etc.

We would like to stress that the result of a reverse engineering process is therefore not necessarily a list of problematic classes or subsystems, even if the identification of possible design defects is a valuable piece of information. The goal of a reverse engineer is to understand the overall structure of the application, to gain a better understanding of the inheritance relationships between classes, and to gain an overview of the methods and the way they are organized. Indeed, we are looking for the bad use as well as the good use of object-oriented design. In that sense, knowing that an inheritance hierarchy is well designed is also valuable information.

Moreover, in a reengineering context the fact that a class may have a design problem does not necessarily imply that the class should be redesigned. Indeed, if a badly designed class or subsystem accomplishes the work it has been assigned to, without having a negative impact on the overall working of the system, there is no point in changing it. However, since reengineers are often not the original developers of the system they are maintaining, being aware of such information is still valuable for getting a better mental model of the system.

4.2.1 A Reverse Engineering Approach Based on Clusters of Polymetric Views

Our approach is based on clusters that loosely group coarse-grained polymetric views depending on the situation encountered by the reverse engineer and the information provided by the view. Each of these clusters is presented in detail in the following sections. We identified four clusters: First Contact, Inheritance Assessment, Candidate Detection, and Class Internal which is discussed in Chapter 5. The views in each cluster provide answers to the questions listed previously, and also answer minor questions that can be derived from those.

In Figure 4.1 we have represented the four clusters of useful views in the context of time. Note that during the process the reverse engineer navigates back and forth from view to view, depending on the encountered symptoms and on the goals of the reverse engineering process. The primary goal is to get an initial understanding of the system which helps the reverse engineer to develop a first mental model of the legacy system [DEME 02].

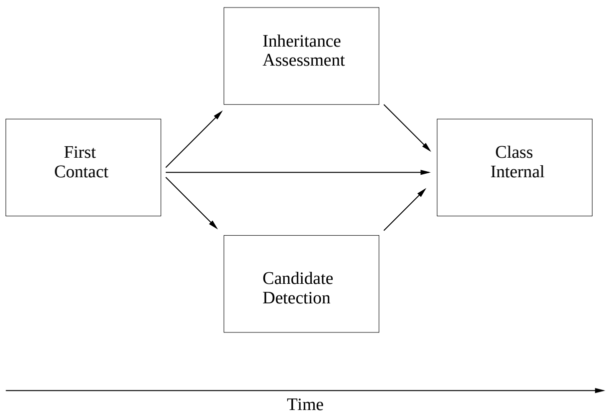


Figure 4.1: Our reverse engineering approach. It is based on 4 clusters whose views are applied at different times during the process.

- **First Contact.** The first thing to do with a subject system is to gain an overview. We would like to know how big and complex the system is and in which way it is structured. The views in this cluster provide answers to the following questions: How big is the system and how is it composed: only of standalone classes, or of some (maybe large) inheritance hierarchies? Is the system composed of many small classes or are there some really big ones? Where in the system do these large classes reside? This cluster contains the views SYSTEM HOTSPOTS, SYSTEM COMPLEXITY, ROOT CLASS DETECTION and IMPLEMENTATION WEIGHT DISTRIBUTION.
- **Inheritance Assessment.** Inheritance is a key aspect of object-oriented programming languages, and thus represents an important perspective from which to understand applications. Inheritance can be used in different ways, for example as pure addition of functionality in the subclasses or as an extension of the functionality provided by the superclasses. The views in this cluster help in the analysis of inheritance and provide answers to the following questions: How are inheritance hierarchies structured and how do they make use of inheritance? Are subclasses merely adding new functionality of redefining the functionality defined in the superclasses? This cluster contains the views INHERITANCE CLASSIFICATION and INHERITANCE CARRIER.
- **Candidate Detection.** One of the primary goals of a reverse engineer is to detect candidates for a more in-depth analysis which may be either cases where further investigation is necessary or where code refactorings are needed. The views in this cluster help in this problem detection process and provide answers to the following questions: Where are the large (small) classes or methods? Are there methods which contain dead code or attributes which are never used? This cluster contains the views DATA STORAGE CLASS DETECTION, METHOD EFFICIENCY CORRELATION, DIRECT ATTRIBUTE ACCESS and METHOD LENGTH DISTRIBUTION.
- **Class Internal.** Understanding classes is a key activity in object-oriented programming, since classes represent the primary abstractions from which applications are built. The main problem of this task is to quickly grasp the purpose of a class and its inner structure. We present and discuss the most important polymetric view in this cluster, the CLASS BLUEPRINT view, in the next chapter. However, note that several of the views belonging to the other clusters can easily be applied on single classes as well, and can thus be found in this chapter.

4.2.2 The Need for a Reverse Engineering Approach

We applied our approach on several large industrial applications ranging from a system of approximately 1.2 Million lines in C++ to a Smalltalk framework of approximately 3000 classes (600 kLOC). During our experiments we were able to quickly gain an overall understanding of the analyzed applications, identify problems and point to classes or subsystems for further investigation.

Moreover we learned that the approach is preferably applied during the first contact with a software system, and provides maximum benefit during the one or two first weeks of the reverse engineering process.

We have already presented a first version of our approach [DUCA 01a], and now present an extended and elaborated version. Ideally such an approach defines which views to apply depending on the short-term and long-term goals of the reverse engineer, what the paths are between the different views, and on what parts of the system the next view should be applied. Such an approach is difficult to elaborate for the following reasons:

- There is no unique or ideal path through the views.
- Different views can be applied at the same stage depending on the current context.
- The decision to use a certain view most of the time depends on some interactions with the currently displayed view.
- The views can be applied to different entities implying some navigation facility between the different views.
- A view displays a system from a certain perspective that emphasizes a particular aspect of the system. However, the view has to be analyzed and the code understood to determine if the details revealed by the view are interesting for further investigation.
- The views are heavily customizable. For instance exchanging two metrics is very easy, but it may result in completely different views. The reverse engineer must steer this process in order to apply useful views and customize those views to get other useful views.

4.3 A Reverse Engineering Scenario

In this section we introduce shortly the case study and then illustrate our reverse engineering approach by presenting and discussing several useful views in detail. We also present the application of the views on the case study. At the end on this section we summarize our findings and present some of the industrial case studies we have performed.

Reporting about a case study is quite difficult without sacrificing the exploratory nature of our approach. Indeed, the idea is that different views provide different yet complementary perspectives on the software. Consequently, a concrete and unambiguous reverse engineering strategy should be to apply the views in some predefined order, but the exact order varies depending on the kind of system at hand and the kind of questions driving the reverse engineering effort. Therefore, readers should read this case study report as one possible use case, keeping in mind that reverse engineers always customize their approach to a particular reverse engineering project.

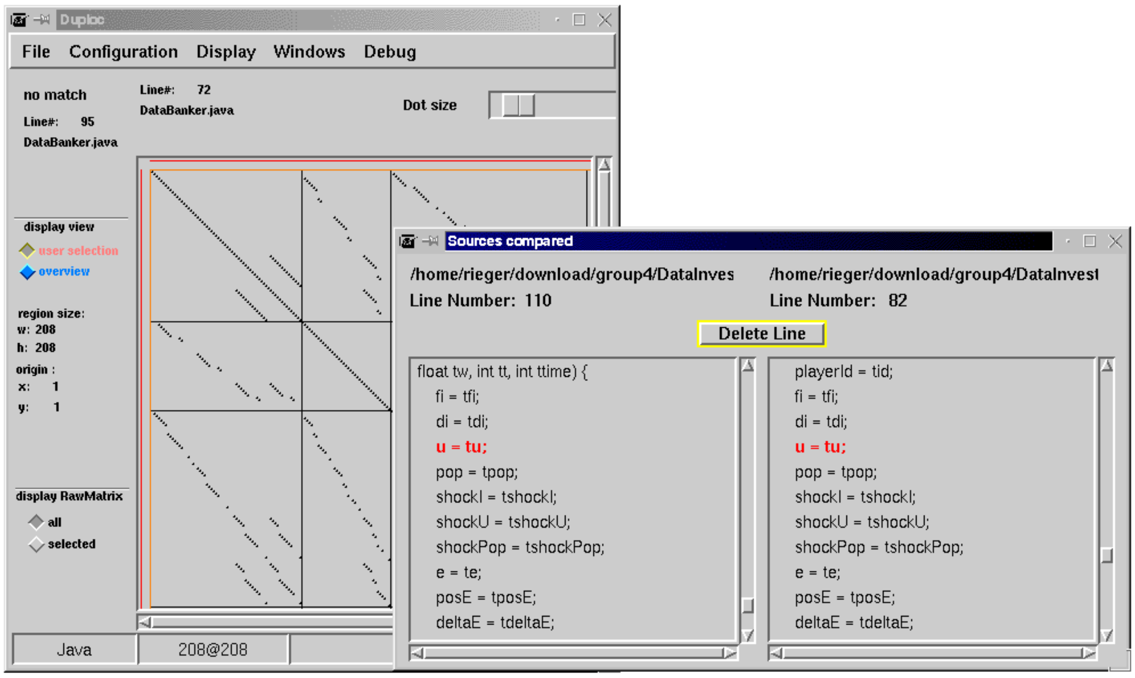
Some Facts about the Case Study. The system we report on is called Duploc (version 2.16a), which is a tool for the detection of duplicated code [DUCA 99]. We have chosen Duploc as case study because it is a freely obtainable application, so that the results presented here can be replicated and verified by others. We have already done a preliminary case study on an older version from 1999 of Duploc [LANZ 99], and are curious to see how Duploc has evolved in the meantime. Duploc has become quite a large system, as we see in Table 4.1, and is thus complex enough to justify a reverse engineering. Duploc detects code duplication by means of a two-dimensional visualization of each line in a matrix, as we see in Figure 4.2.

Note that the numbers presented in Table 4.1 include metaclasses and stub classes.

- Metaclasses are included because in Smalltalk we consider a metaclass as being a different entity than a class. In Smalltalk metaclasses encode class behavior, in simple words we could say that metaclasses implement the *static* methods. Note that this is not completely correct, as metaclasses have much more power, but such a discussion is out of the scope of this thesis.

Entities	**Amount**
Classes	758
Methods	5493
Attributes	794
Relationships	**Amount**
Inheritance Relationships	634
Method Invocations	19327
Attribute Accesses	30682

Table 4.1: An overview of the size of the Duploc case study.



no match
Line#: 95
DataBanker.java

Line#: 72
DataBanker.java

region size:
w: 208
h: 208
origin :
x: 1
y: 1

Java | 208@208

/home/rieger/download/group4/DataInves
Line Number: 110

```
float tw, int tt, int ttime) {
    fi = tfi;
    di = tdi;
    u = tu;
    pop = tpop;
    shockI = tshockI;
    shockU = tshockU;
    shockPop = tshockPop;
    e = te;
    posE = tposE;
    deltaE = tdeltaE;
```

/home/rieger/download/group4/DataInvest
Line Number: 82

```
    playerId = tid;
    fi = tfi;
    di = tdi;
    u = tu;
    pop = tpop;
    shockI = tshockI;
    shockU = tshockU;
    shockPop = tshockPop;
    e = te;
    posE = tposE;
    deltaE = tdeltaE;
```

Figure 4.2: A screenshot from the Duploc case study.

- Stub Classes are included for reasons of completeness: We have selected all classes which belong to the Duploc application, but since the Duploc classes also inherit from other classes or reference them, we include these other classes as empty stub classes.

Therefore the actual number of classes in the Duploc application is smaller, namely 317. The numbers in the table are derived like this: 758 = 317 (classes) + 317 (metaclasses) + 62 (stub classes) + 62 (stub metaclasses). The other metrics in the table are however correct, since stub classes are considered empty in our model.

4.3.1 Reverse Engineering a System

Reverse engineering a system is a non-linear procedure and is difficult to present as a sequential text. For reasons of simplicity we discuss the views of each of the clusters, show their application to the case study and put them into relation according to our approach presented in Section 4.2, as well as depending on the situations encountered during the reverse engineering of Duploc.

In the remainder of this section we present several of these views in detail. Every view comes with a small description which includes the name of the view, the used layout, the scope of the view, the entities which it visualizes, and a list of the used metrics, and an optional sort criterion according to which the nodes have been sorted, and which in some cases influences the position of the nodes. Moreover, we provide a detailed description of the view which includes the following: First a short presentation of the idea behind the view, then a list of the symptoms the viewer should watch out for, and finally possible variations of a view and their effects. Interspersed in the text we also point out other related views which could be applied. Then we apply the views on the case study and discuss the findings.

4.3.2 First Contact Views

The main purpose of these views is to obtain a first overview of a subject system. This cluster contains the following views:

1. SYSTEM HOTSPOTS VIEW
2. SYSTEM COMPLEXITY VIEW
3. ROOT CLASS DETECTION VIEW
4. IMPLEMENTATION WEIGHT DISTRIBUTION VIEW

SYSTEM HOTSPOTS VIEW

Layout	Checker	
Nodes	Classes	
Edges	-	
Scope	Full System	
Metrics		
Size	NOA (number of attributes)	NOM (number of methods)
Color	-	
Position	-	-
Sort	Width	
Example	Figure 4.3	

Description

This simple view helps to identify large and small classes and scales up to very large systems. It relates the number of methods with the number of attributes of a class. The nodes are sorted according to the number of methods, which makes the identification of outliers easy.

Reverse Engineering Goals

This view gives a general impression of the system in terms of overall size (how many nodes are there?) and in terms of the size of the classes (are there any really large nodes and how many large nodes are there?)

Symptoms

1. Large nodes represent voluminous classes that may be further investigated.
2. Tall, narrow nodes represent classes which define many methods and few or no attributes.
3. Wide nodes are classes with many attributes. When such nodes show a 1:2 width-height ratio it may represent a class whose main purpose is to be a data structure implementing mostly accessor methods.

Variations

1. If we use the lines of code (WLOC) or the number of methods (NOM), as we see in Figure 4.3 for rendering both the width and height of the nodes, we obtain a slightly different view which helps to assess the whole system in terms of raw measure: are there any big classes and how big are they actually?
2. In the case of Smalltalk classes, we can color metaclasses differently and check how they distribute themselves across the display. Should there now be large, colored nodes at the bottom of the display, it may be a sign that these metaclasses have too many responsibilities or that they function facades or as bridges to other classes [GAMM 95].
3. Further evidence can be gained from the color, which can be used to reflect the number of lines of code of a class. Should a tall class have a light color it means that the class contains mostly short methods.

Scenario

In Figure 4.3 we see all the Duploc classes. The classes in the bottom row contain more than 100 methods and should be further investigated. They are *DuplocPresentationModelController (107 methods),*

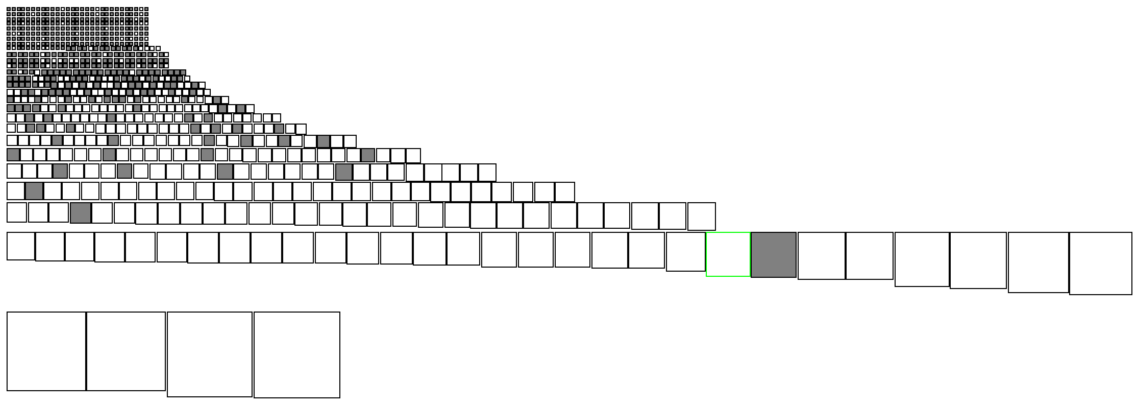


Figure 4.3: A SYSTEM HOTSPOTS view (Variation 1+2) of Duploc. The nodes represent all the classes, while the size of the nodes represent the number of methods they define. The grey nodes represent metaclasses.

RawMatrix (107), DuplocSmalltalkRepository (116) and DuplocApplication (117 methods). We have colored the nodes representing metaclasses with grey. Note the bottom-most grey node which is the metaclass *DuplocGlobals* with 59 methods. This class, as suggested as well by the name, is a holder for global values. However, instead of using the metaclass, one suggestion to the developer is to apply the singleton design pattern instead [GAMM 95].

This view shows that Duploc is a system of more than 300 classes, where the largest classes contain more than 100 methods. It also shows an impressive number of very small classes implementing few methods.

SYSTEM COMPLEXITY VIEW

Layout	Tree	
Nodes	Classes	
Edges	Inheritance	
Scope	Full System	
Metrics		
Size	NOA (number of attributes)	NOM (number of methods)
Color	WLOC (lines of code)	
Position	-	-
Sort	-	
Example	Figure 4.4	

Description

This view is based on the inheritance hierarchies of a subject system and gives clues on its complexity and structure. For very large systems it is advisable to apply this view first on subsystems, as it takes quite a lot of screen space. The goal of this view is to classify inheritance hierarchies in terms of the functionality they represent in a subject system. If we want to understand the inner working at a technical level of inheritance hierarchies we apply the views of the inheritance assessment cluster.

Reverse Engineering Goals

The view helps to identify and locate the important inheritance hierarchies, but also shows whether there are large classes not part of a hierarchy (possibly god classes [RIEL 96]). Like all views of the first contact cluster it also answers the question about the size of the subject system. Moreover, it helps to detect exceptional classes in terms of number of methods (tall nodes) or number of attributes (wide nodes).

Symptoms

1. Tall, narrow nodes represent classes with few attributes and many methods. When such nodes appear within a hierarchy, applying the INHERITANCE CLASSIFICATION view or the INHERITANCE CARRIER view helps to qualify the semantics of the inheritance relationships in which the classes are involved, as we see in the discussion of those views.

2. Deep or large hierarchies are definitively subsets of the system on which the views of the inheritance assessment cluster should be applied to refine their understanding.

3. Large, standalone nodes represent classes with many attributes and methods without subclasses. It may be worth looking at the internal structure of the class to learn if the class is well structured or if it could be decomposed or reorganized.

Variations

1. A valuable variation of this view is the INHERITANCE CLASSIFICATION view discussed later in this chapter.

Scenario

We perform a manual preprocessing which consists of removing the class *Object*, which is the root class of the Smalltalk language. We do this in order to focus on the use of inheritance within Duploc: Since many classes inherit directly from *Object* this view would be distorted if we included it in our view. We see the resulting SYSTEM COMPLEXITY view in Figure 4.4. We can see now that Duploc is in fact mainly composed of classes which are not organized in inheritance hierarchies. Indeed, there are some very

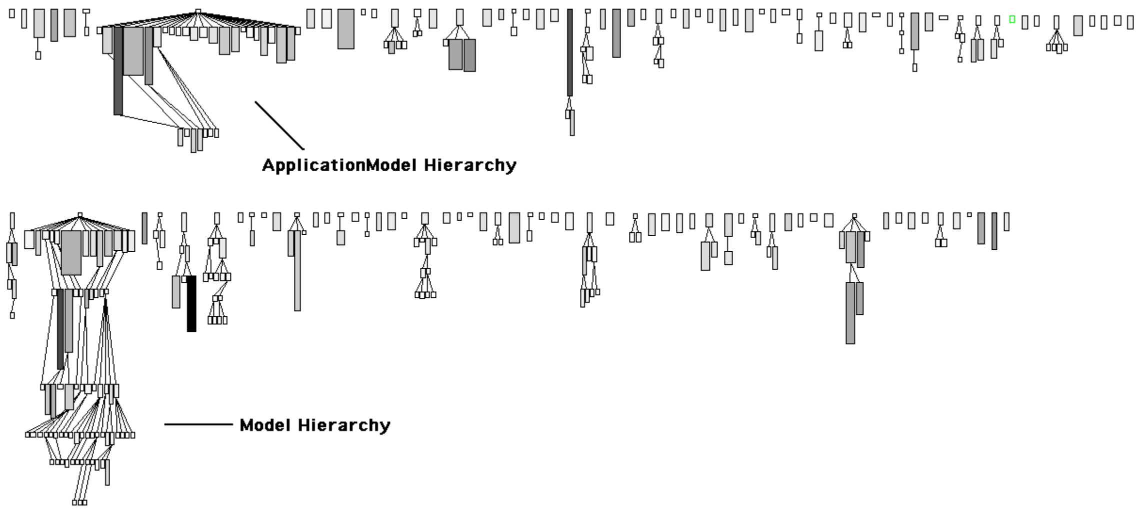


Figure 4.4: A SYSTEM COMPLEXITY view on Duploc. The nodes represent the classes, while the edges represent inheritance relationships. As metrics we use the number of attributes (NOA) for the width, the number of methods (NOM) for the height and the number of lines of code (WLOC) for the color.

large classes which do not have subclasses. The largest inheritance hierarchies are five and six levels deep. Noteworthy hierarchies seem to be the ones with the following root classes: *AbstractPresentationModelControllerState*, *AbstractPresentationModelViewState*, *DuplocSourceLocation*. By manually inspecting the first one, with the root class *AbstractPresentationModelControllerState* with 31 descendants, we infer that it seems to be the application of the *state* design pattern [GAMM 95, ALPE 98] for the controller part of an MVC pattern. Such a complex hierarchy within Duploc is necessary, since Duploc does not make any use of advanced graphical frameworks, but uses the basic standard VisualWorks GUI framework. Following this track of investigation we look for the other signs of the MVC pattern and find a hierarchy with *AbstractPresentationModelViewState* as root class with 12 descendants, which seems to constitute the view part of the MVC pattern.

This view shows that Duploc consists of several very small hierarchies composed of small classes and two bigger hierarchies, where one represents the domain model of Duploc (the Model hierarchy), and the other one contains all GUI-related classes (the ApplicationModel hierarchy).

ROOT CLASS DETECTION VIEW

Layout	Scatterplot	
Nodes	Classes	
Edges	-	
Scope	Full System	
Metrics		
Size	-	-
Color	-	
Position	WNOC (number of descendants)	NOC (number of subclasses)
Sort	-	
Example	Figure 4.5	

Description

This view helps in identifying the *roots* of inheritance hierarchies by putting in relation the number of direct subclasses with the number of descendant classes, *i.e.*, all direct and indirect subclasses. This view is mainly useful for very large systems, where a view like SYSTEM COMPLEXITY may have problems to represent the whole system on one screen.

Reverse Engineering Goals

This view helps to identify the roots of the large inheritance hierarchies, and depending on the number of nodes that are more to the right also gives an impression of the overall use of inheritance in the system.

Symptoms

1. Nodes to the right of the display have many descendants.
2. Nodes to the bottom of the display have many direct subclasses.

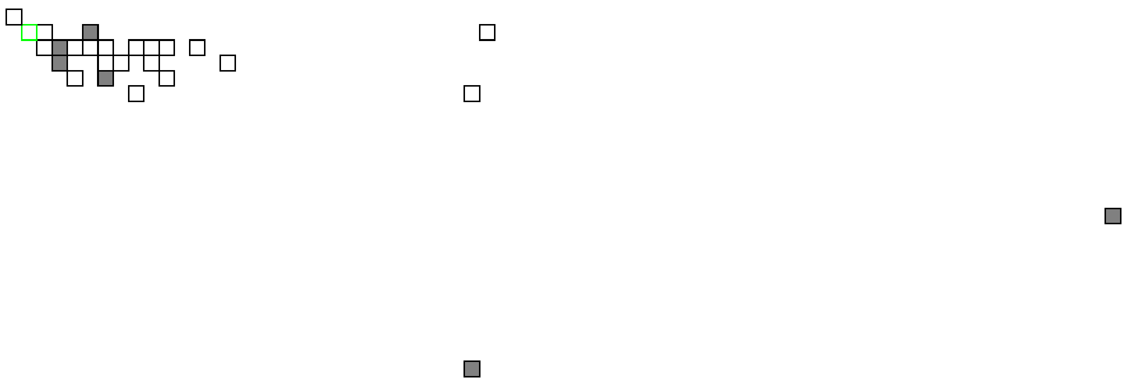


Figure 4.5: A ROOT CLASS DETECTION view (Variation 1) of Duploc. The horizontal position of the class nodes renders the number of descendants of each class, while the vertical position renders the number of immediate subclasses of each class.

Variations

1. If we use a different color to denote abstract classes, we are able to see whether the top-level classes (the nodes to the right and bottom of the display) are declared as such, as we would expect.

Scenario

We expect that this view will confirm our findings obtained with the SYSTEM COMPLEXITY view. Indeed, as we see in Figure 4.5 there are two classes which stand out from the others, *AbstractPresentationModelControllerState, PMCS* with 31 and 30 descendants, The latter is the sole direct subclass of the former. We have colored the classes in case they are abstract, and see that *PMCS* is not abstract, although at the top of the largest inheritance hierarchy. After a short verification, we see that in fact *PMCS* has in turn abstract subclasses. Note that this is possible – although not advisable – in Smalltalk, because there are no constructs at the language level which declare a class as being abstract, like in the case of C++ and Java. In Smalltalk the abstractness of a class is implied by declaring one of its methods as abstract. However, this is an idiom and not enforced at a language level.

This view shows that Duploc has only two major inheritance hierarchies, as is denoted by only two outlying nodes.

IMPLEMENTATION WEIGHT DISTRIBUTION VIEW

Layout	Histogram	
Nodes	Classes	
Edges	-	
Scope	Full System	
Metrics		
Size	NOM (number of methods)	-
Color	HNL (hierarchy nesting level)	
Position	-	NOM (number of methods)
Sort	-	
Example	Figure 4.6	

Description

This view gives a qualitative overview of complete systems by categorizing them as *top-heavy, bottom-heavy or mixed*. The histogram lays out the classes according to their size, while the color metric reflects the metric HNL, i.e., the depth of a class within an inheritance hierarchy. Note that this view must be looked at in general terms: the purpose is not to inspect and qualify each single node, but to look at the display as a whole.

Reverse Engineering Goals

This view helps to gain a general impression of the system in terms of the use of inheritance and the size of the classes.

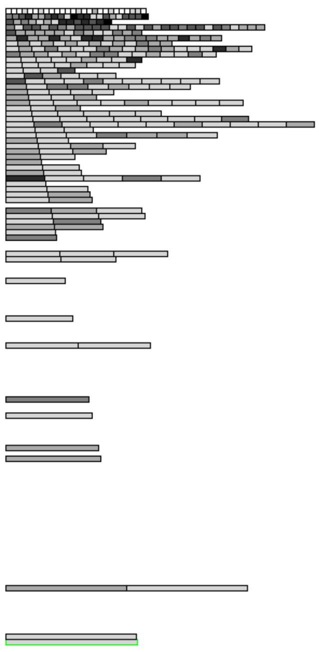


Figure 4.6: An IMPLEMENTATION WEIGHT DISTRIBUTION view of Duploc. The width and vertical position of the class nodes represents the number of methods, while the color represents the hierarchy nesting level, e.g., the depth of the classes within an inheritance hierarchy.

Symptoms

1. Dark nodes at the bottom of the display represent large classes deep within a hierarchy. Such systems are bottom-heavy, because these symptoms reveal that the larger classes in the system are mainly at the bottom of inheritance hierarchies, which may be a result of abuse of abstraction mechanisms.

2. Light nodes at the bottom of the display reveal that the large classes in the system reside in the higher levels of the inheritance hierarchies, which may be also a problem, as subclassing large and complex classes can be difficult.

3. The light and dark nodes are evenly distributed across the display, which classifies the system as being mixed.

Variations

None

Scenario

We gather from Figure 4.6 that according to the definition of this view, Duploc is a top-heavy system, *i.e.*, the main weight of its implementation resides in the top-level classes.

This view shows that Duploc does not have very large classes, because the histogram would have been taller then. It also shows that there are few large classes towards the bottom of the inheritance hierarchies. This is also due to the fact that Duploc consists mostly of small inheritance hierarchies.

4.3.3 Inheritance Assessment Views

These views help to understand and qualify the use of inheritance in a subject system. This cluster contains the following views:

1. INHERITANCE CLASSIFICATION VIEW
2. INHERITANCE CARRIER VIEW
3. INTERMEDIATE ABSTRACT VIEW

INHERITANCE CLASSIFICATION VIEW

Layout	Tree	
Nodes	Classes	
Edges	Inheritance	
Scope	Subsystem	
Metrics		
Size	NMA (number of methods added)	NMO (number of methods overridden)
Color	NME (number of methods extended)	
Position	-	-
Sort	-	
Example	Figure 4.7	

Description

This view qualifies the inheritance relationships by displaying the amount of added methods relative to the number of overridden or extended methods. By extended methods we mean methods which contain a super call to a method with the same signature defined in one of the superclasses.

Reverse Engineering Goals

This view helps to understand the use of inheritance in class hierarchies, and reveals whether a hierarchy is built on code reuse through extending and overriding methods, or on mere addition of functionality.

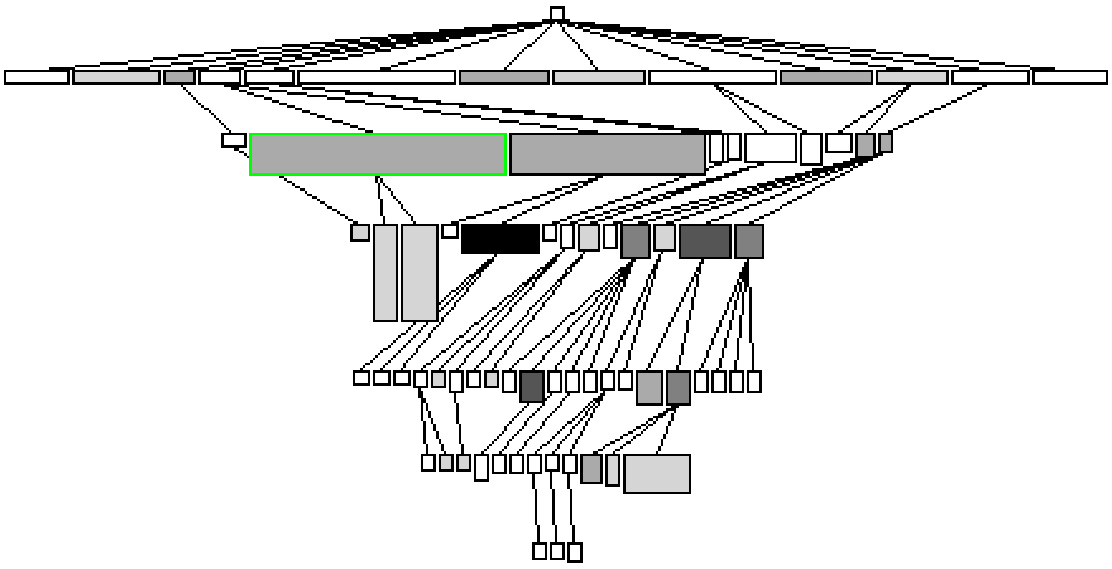


Figure 4.7: An INHERITANCE CLASSIFICATION view of the *Model* hierarchy in Duploc. The width and height of the class nodes represents the number of added methods and the number of overridden methods, while the color represents the number of extended methods.

Symptoms

1. Flat, light nodes represent classes where a lot of methods have been added but where few methods have been overridden or extended. In this case the semantic of the inheritance relationship is an

addition of functionality by the subclasses.

2. Tall, possibly darker nodes represent classes where a lot of methods have been overridden and/or extended. They may represent classes that have specialized hook methods [GAMM 95]. If the nodes are dark, it means that many methods have been extended, which hints at a higher degree of reuse of functionality.

Variations

None

Scenario

We have selected only one hierarchy, the one indicated as the *Model* hierarchy in Figure 4.4, to demonstrate the application of this view. We see in Figure 4.7 that the *Model* hierarchy is mainly composed of flat, lightly colored nodes: these classes mainly add functionality (denoted by their width) without really overriding or extending functionality defined in the superclasses. We also see there are some exceptions: the subclasses of the two widest class nodes (*RawMatrix* and *AbstractRawSubMatrix*) with 96 and 72 added methods define several methods which are then overridden or extended by their subclasses. For example the two subclasses (*SymmetricRawMatrix* and *AsymmetricRawMatrix*) of *RawMatrix* heavily override functionality, as is indicated by their tall, narrow shape: both override 33 methods and add only 4, respectively 9, methods.

This view applied on the *Model* hierarchy of Duploc shows that inheritance is used in both senses, *e.g.*, there are parts of this hierarchy where classes add new methods, and other parts where classes reuse and complement the functionality defined in their superclasses by extending or overriding methods.

INHERITANCE CARRIER VIEW

Layout	Tree	
Nodes	Classes	
Edges	Inheritance	
Scope	Subsystem	
Metrics		
Size	WNOC (number of descendants)	NOM (number of methods)
Color	WNOC (number of descendants)	
Position	-	-
Sort	-	
Example	Figure 4.8	

Description

This view helps to detect classes with a certain impact on their subclasses in terms of functionality, *i.e.*, it helps us see which classes transmit the most functionality to their subclasses.

Reverse Engineering Goals

The goal of this view is to refine the understanding of a class hierarchy and see which classes have the most impact on their subclasses.

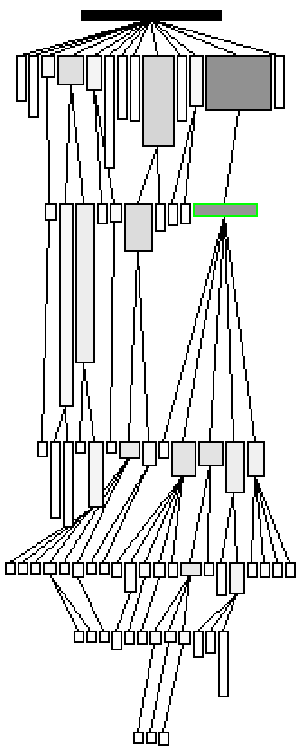


Figure 4.8: An INHERITANCE CARRIER view of one hierarchy in Duploc. The width and the color of the class nodes represents the number of descendants, while the height represents the number of methods.

Symptoms

1. Tall, dark nodes represent classes that define a lot of behavior and have many descendants. Therefore these classes have a certain importance for the (sub)system in question.
2. Flat, light nodes represent classes with little behavior and few descendants.
3. Flat, dark nodes represent classes with little behavior and many descendants. They can be the ideal place to put code factored out from the subclasses.

Variations

None

Scenario

Figure 4.8 shows this view for the *Model* hierarchy. It shows that the classes which are carrying the weight of the implementation in this hierarchy are first of all the classes *AbstractPresentationModelControllerState* and *PMCS*, where the latter is the sole subclass of the former. These classes are emphasized in this view because of their darker color.

This view helped us to identify in the largest class hierarchy of Duploc those two classes which have the most impact on the subclasses of this hierarchy and therefore constitute important information about this hierarchy.

INTERMEDIATE ABSTRACT VIEW

Layout	Tree	
Nodes	Classes	
Edges	Inheritance	
Scope	Subsystem	
Metrics		
Size	NOM (number of methods)	NMA (number of methods added)
Color	NOC (number of subclasses)	
Position	-	-
Sort	-	
Example	-	

Description

This view identifies classes that are nearly empty in the middle of inheritance hierarchies. It uses the number of subclasses (NOC) as color metric, while for the size metrics we use the number of methods (NOM) and the number of methods which have been added compared to the superclass (NMA). Such classes are of some interest, because functionality defined in their subclasses can be pushed up into the intermediate abstract classes, in order to prevent or reduce code duplication.

Reverse Engineering Goals

This view helps to locate within inheritance hierarchies classes which are nearly empty and therefore constitute places where one could factor code out from their subclasses into those intermediate classes.

Symptoms

1. Flat, dark nodes represent classes in which few or no methods have been added, but which have many direct subclasses. Further investigation may reveal that these classes implement some abstract behavior.

Variations

1. Instead of NOC, one could also use WNOC (the number of descendants) for the color, although in large hierarchies the noise generated by other classes may obfuscate the detection of an intermediate abstract class.

Scenario

This view did not yield results in the case of Duploc, *i.e.*, Duploc does not contain an intermediate abstract class.

4.3.4 Candidate Detection Views

These views help to identify candidates for further inspection, *e.g.*, interesting software artifacts like classes, methods, and attributes in terms of size, complexity, and other particularities. Often these candidates represent places where it is useful to apply refactorings. This cluster contains the following views:

1. DATA STORAGE CLASS DETECTION VIEW
2. METHOD EFFICIENCY CORRELATION VIEW
3. DIRECT ATTRIBUTE ACCESS VIEW
4. METHOD LENGTH DISTRIBUTION VIEW

DATA STORAGE CLASS DETECTION VIEW

Layout	Stapled	
Nodes	Classes	
Edges	-	
Scope	Subsystem	
Metrics		
Size	NOM (number of methods)	WLOC (lines of code)
Color	NOM (number of methods)	
Position	-	-
Sort	Width	
Example	Figure 4.9	

Description

This view relates the number of methods (NOM) with the lines of code (WLOC) of classes and interprets this information in the context of a subsystem or small system. Ideally this view should return a nice staircase pattern from left to right, since the nodes are sorted according to the first metric and the two metrics are related. Note that this view works in any setting, i.e., since it puts two values in relation it doesn't matter how big the actual measurements are.

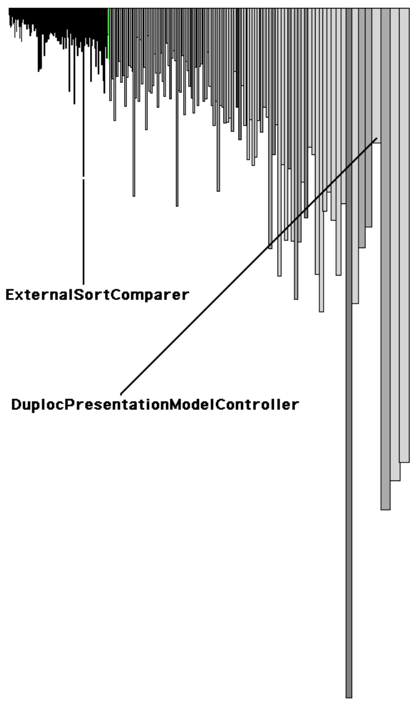


Figure 4.9: A DATA STORAGE CLASS DETECTION view on the largest classes in terms of number of methods of Duploc. The color and height metrics represents the number of lines of code of each class, while the width represents the number of methods. The nodes are sorted according to their width.

Reverse Engineering Goals

This view helps to detect data storage classes, *e.g.*, classes which mainly hold data and do not have complex behavior. Such classes are normally heavily used by other classes which want to access the information contained in the data storage classes. Moreover, it also helps to detect classes with an exceptional average length of the methods compared to the rest of the subject system.

Symptoms

1. The staircase effect is broken by nodes which are too tall. These represent classes which have long methods compared to the classes which comply with the staircase pattern.
2. The staircase pattern is broken by nodes which are too short. These classes, given a certain number of methods, do not have the expected length in terms of lines of code. Such classes are often data storage classes, *i.e.*, classes which have short, simple methods, possibly only accessor methods. Data storage classes may point to sets of coupled classes being brittle to changes.

Variations

1. To enhance the detection of data storage classes we can use the number of attributes (NOA) as color metric, because data storage classes often have many attributes.

Scenario

We see in Figure 4.9 that the fourth class from the right, *DuplocPresentationModelController* is very short (265 line of code) compared to the great number of methods (107) indicated by the position on the right. Upon closer inspection we see that the class contains dozens of one-line methods which return constant values. We also see the inverse case for the first tall class on the left named *ExternalSortComparer* which contains 12 methods for a total length of 330 lines. Through a manual verification by code reading we have indeed assessed that this class contains methods which can be refactored by splitting them up in smaller, more reusable pieces.

This view helped us to identify several candidates which are possibly data storage classes, and manual verifications have proved us right in most of the cases. Moreover it also helped us to identify classes with overly long methods which are candidates for method splitting refactorings.

METHOD EFFICIENCY CORRELATION VIEW

Layout	Correlation	
Nodes	Methods	
Edges	-	
Scope	Full System	
Metrics		
Size	-	-
Color	-	
Position	LOC (lines of code)	NOS (number of statements)
Sort	-	
Example	Figure 4.10	

Description

This very scalable view shows all methods using a scatterplot layout with the lines of code (LOC) and the number of statements (NOS) as position metrics. As the two metrics are related (each line may contain statements) we end up with a display of all methods, many of which align themselves along a certain correlation axis.

Reverse Engineering Goals

The goal of this view is to help detect (1) overly long methods, (2) methods with dead code, (3) badly formatted methods.

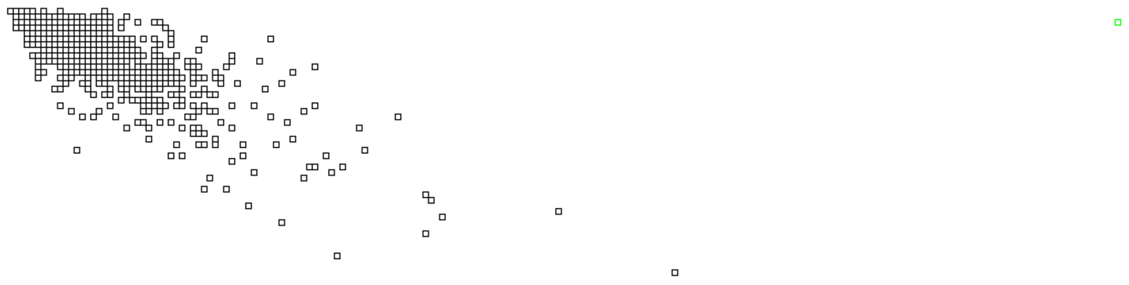


Figure 4.10: A METHOD EFFICIENCY CORRELATION view of Duploc. As horizontal position metric we use the lines of code, while for vertical metrics we use the number of statements.

Symptoms

1. Nodes to the right of the display represent long methods and should be further investigated as candidates for split method refactorings [FOWL 99, BECK 97].
2. Nodes to the very left and top of the display represent empty methods.
3. Nodes to the top of the display, but not necessarily to the left, represent methods containing only commented lines and thus possibly represent dead code.
4. Nodes to the left and more to the bottom of the display represent methods which are probably hard to read, as they contain several statements on each line. In this case one should check whether there are formatting rules within the application which are being violated.

Variations

1. This view can be enriched using size metrics as well. One useful variation is using the number of parameters (NOP) for the size of the nodes, which reveals not only long methods but methods with many input parameters as well.

Scenario

We can see in Figure 4.10 how well this view scales up: the figure shows nearly 5000 of Duploc's methods. Several method nodes seem to be good candidates for further investigations. All the methods longer than a certain number of lines (for example 30 or 50, depending on the average length of methods in the subject system) should be inspected. Note in this regard that the average length of Smalltalk methods is around 7 lines [KLIM 96]. We can also see that there are many methods at the top of the display which therefore do not contain many statements. Upon closer inspection we can see this is partly due to code which is commented out (in some cases dead code), partly this is also due to very long comments written by the developer to explain what the methods are actually doing. Another insight which can come from this view is a general assessment of the system. We have seen that the methods tend to align themselves along a certain correlation axis. Depending on the age of the system the axis changes its angle: methods are written and corrected all the time, and slowly get messy with many statements on few lines. In this regard Duploc can still be considered a young system.

This view helped us to detect several methods which were too long compared to the rest of the system, as well as dead code methods, *e.g.*, methods with commented bodies. The actual concrete result of this view is usually to produce a list of candidates which can be used for (1) documentation, and (2) future inspection and eventually application of refactorings.

DIRECT ATTRIBUTE ACCESS VIEW

Layout	Checker	
Nodes	Attributes	
Edges	-	
Scope	Full System	
Metrics		
Size	NAA (number of total direct accesses)	NAA (number of total direct accesses
Color	NAA (number of total direct accesses)	
Position	-	-
Sort	Width	
Example	Figure 4.11	

Description

This view uses for all visualized attribute nodes (scales up to complete systems) the number of direct accesses (NAA) for the width, height and color of each node, and sorts the nodes according to this metric. It can be used to assess the usage of attributes in a system, as well as for the detection of unused attributes.

Reverse Engineering Goals

The goal of this view is to get an impression of how attributes are accessed and used. It can answer questions about the type of accesses (direct/indirect over accessor methods) and helps to detect dead, *e.g.*, unused attributes.

Symptoms

1. Small nodes at the top of the display represent attributes which are never accessed and may point to dead code.
2. Large, dark nodes at the bottom point to attributes which heavily directly accessed, which may lead to problems, in case the internal implementation changes. For such nodes one should also check whether accessor methods have been defined, and if yes why they are not always being used.

Variations

1. Instead of using as size and color metric the number of direct global accesses, we can use either the number of accesses via accessor methods to reveal how heavily these accessor methods are actually used.
2. We can use as size and color metric the number of direct accesses by subclasses, in order to reveal coupling aspects of classes within inheritance hierarchies.
3. We can use the number of local accesses (NLA) (from within the class where the attribute resides) for the width and the number of global accesses (NGA) (from outside of the class) for the height. Normally the attributes rendered like this should be as flat as possible, and in cases where this does not apply, a deeper inspection could be useful, since tall, narrow nodes represent attributes which are heavily accessed from outside of its defining class by means of direct accesses.

Scenario

In Figure 4.11 we use a slight variation of the regular view definition and render for the width and the height the number of local, respectively the number of non-local accesses, while the color renders the total number of direct accesses. We see that Duploc uses a considerable number of attributes. The top row contains 11 attributes which are never accessed, and can therefore be removed. The bottom row contains

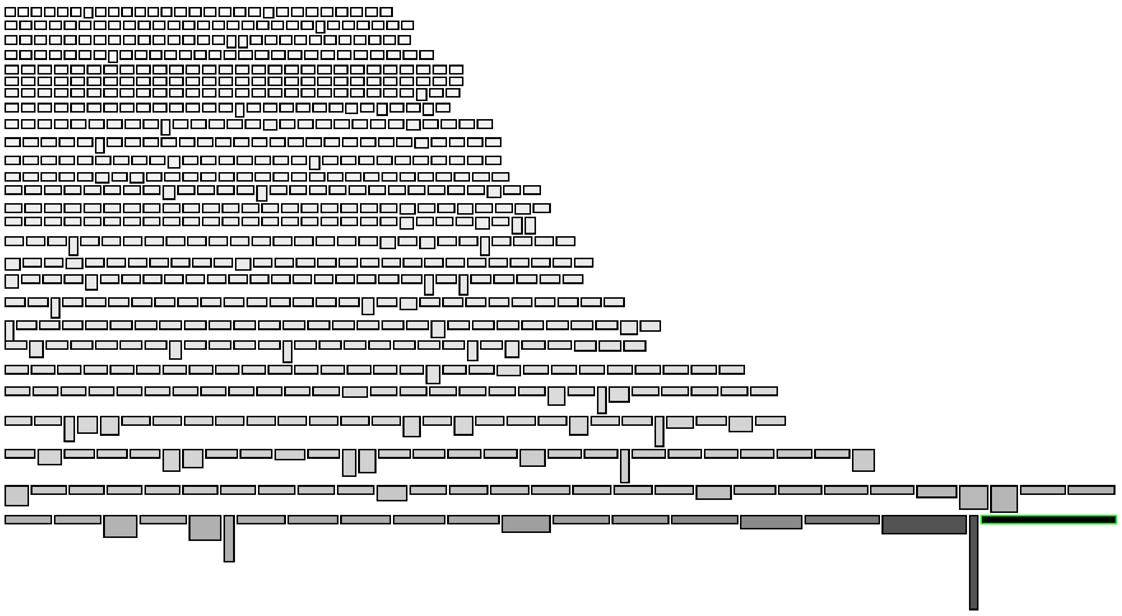


Figure 4.11: A DIRECT ATTRIBUTE ACCESS view (Variation 3) of Duploc. The width of each attribute node represents the number of direct local accesses from within its defining class (NLA). The height of each node represents the number of accesses from outside of its class (NGA), while the color represents the number of total direct accesses. The nodes are sorted according to the color metrics.

the most heavily accessed attributes. For example the attribute *bvcm* belonging to class *BinValueColorerInterface* is directly accessed 77 times. Upon closer inspection we see that in fact the class defines accessor methods, but they are not consistently used, which may be risky. Note also the tall, narrow attribute node at the bottom of this view. This attribute is heavily accessed directly from outside of its containing class. In such a case we suggest to define accessor methods and invoke them instead of directly accessing the attribute.

We have seen that this view helped to detect unused attributes in Duploc, as well as heavily accessed attributes, and also identified attributes which methods access in an inconsistent style, *e.g.*, sometimes directly, sometimes indirectly over the accessor methods.

METHOD LENGTH DISTRIBUTION VIEW

Layout	Histogram	
Nodes	Methods	
Edges	-	
Scope	Class(es)	
Metrics		
Size	LOC (lines of code)	-
Color	-	
Position	-	LOC (lines of code)
Sort	Width	
Example	Figure 4.12, Figure 4.13	

Description

This view is mainly applied on single classes or small groups of classes. It reveals the shape of a class in terms of the distribution of the methods according to their length. This view is usually applied in parallel to other views.

Reverse Engineering Goals

This view, mainly applied to single classes or small groups of classes reveals which methods are the longest.

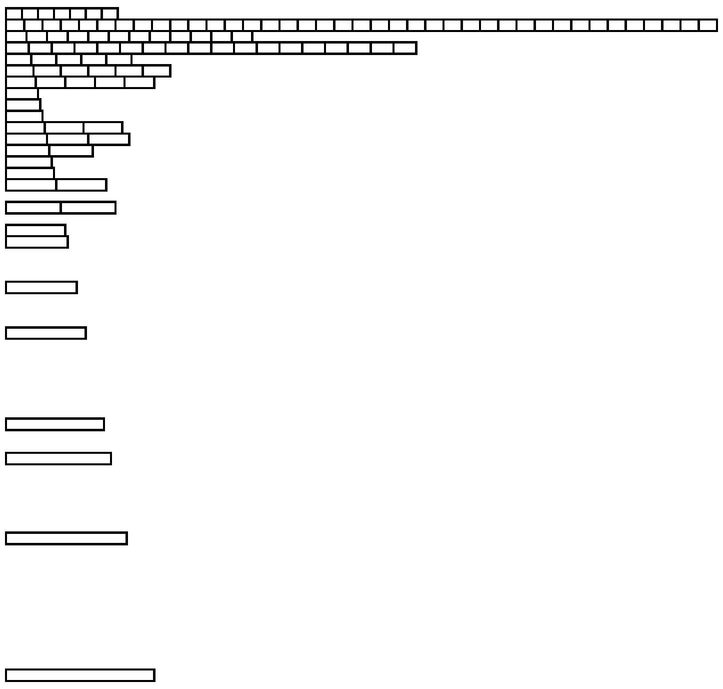


Figure 4.12: A METHOD LENGTH DISTRIBUTION view of the class DuplocApplication. The width of each method node and the vertical position is represented by the number of lines (LOC)

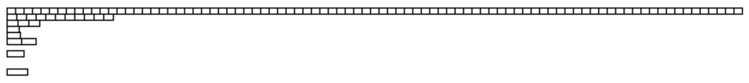


Figure 4.13: A METHOD LENGTH DISTRIBUTION view of the class DuplocPresentationModelController. The width of each method node and the vertical position is represented by the number of lines (LOC)

Symptoms

1. Wide nodes at the bottom represent the longest methods, and are candidates for refactorings.
2. Nodes which are standing farther away represent exceptional cases. For example if all the methods of a class have 10 or less lines of code, and one method has 25 lines of code it will separate itself from the other methods in the view.
3. Small nodes at the very top represent empty methods.

Variations

1. For the color we can also use the number of invocations (NI) or the number of statements (NOS). As both metrics are in relation with the length of a method, this variation helps to detect dead code, in case we have light nodes at the bottom of the display.

Scenario

We show two example classes of Duploc. From the first example class, DuplocApplication, shown in Figure 4.12 we gather that in this class most of the methods are very short, with a few very long methods compared to the average length of the methods in this class. These exceptions could be closer inspected for a possible method splitting refactoring. A stark counterexample is provided by the class DuplocPresentationModelController, shown in Figure 4.13, which shows that this class contains only very short methods.

This view helped us to see that the two classes we showed look quite different regarding the length of the methods they contain.

4.3.5 Case Study Evaluation

Our approach provides us with an initial understanding of the case study and helps us to identify some of the key classes without having to dive into the details. Indeed, one of the major problems with large systems is to get an overview and some initial understanding without getting lost in their intrinsic complexity. The reverse engineering approach based on clusters of views helps us stay focused at the different levels of understanding we want to gain. We cannot present all the results we obtained during this case study, as this would go beyond the scope of this chapter. We rather limit ourselves to draw some specific conclusions from the major findings obtained during this case study, and some general conclusions from other case studies we have performed.

First Contact Views. The views in this cluster help us to get an impression of the size and structure of a system, and in more detail to see how a system's major hierarchies are composed and where larger classes are located. In the present case, we have seen that Duploc is composed of several standalone classes, and that a major part of Duploc is dedicated to the management of the graphical user interface. A first list of prominent classes and hierarchies of the system is useful to get an orientation. Especially on very large case studies, this cluster's views help to obtain results quickly. This is important for the reverse engineer in order to decide in which direction the reverse engineering process must go.

Inheritance Assessment Views. The views in this cluster are useful for the understanding of the complex mechanisms which come with inheritance. We can classify inheritance relationships and detect important classes in large hierarchies. Especially for larger hierarchies, which however this case study did not contain, this cluster's views reduce the time to understand complete inheritance hierarchies. In one special case, we reverse engineered a system which contained very large inheritance hierarchies, with several hundreds of classes and where in one case the root class, had 97 direct subclasses. The views obtained after visualizing this hierarchy led us to coin the term *flying saucer* hierarchy, because of its very flat shape.

Candidate Detection Views. The views in this cluster help us to identify many candidates for closer examination and possibly the application of refactorings. The problem with those candidates is that their number can be great. In one industrial case study we presented a list of all methods longer than 100 lines. The list contained several dozens of methods. Since the system was written in Smalltalk, where the average length of a method is ca. 6 lines, the chief developer asked us to generate a list with all methods being longer than 20 lines instead, which resulted in a list with several hundreds of methods. This little anecdote shows the dangers of the views contained in this cluster: the reverse engineer can easily produce long lists of suspicious code fragments, but the usefulness of such an approach is doubtful: in the end it is the software company that decides on which parts of their system they want to spend time and money for reengineering. In the case of Duploc, it is difficult to present the results in detail, because all the detected candidates must be examined, and this would go far beyond the scope of this chapter.

4.3.6 Industrial Experiences

We have validated our approach in several academic and industrial experiences, some of which we list in detail in Table 4.3.6. However, due to non-disclosure agreements with the industrial partners, we cannot deliver a detailed report on those experiences.

Members of our team went to work on industrial applications in a "let's see what we can tell them about their system" way. The common point about these experiences was that the subject systems were of considerable size and that there were narrow time constraints. This led us to mainly get an understanding of the system and produce overviews. We were also able to point out potential design problems and on the medium-sized case study we even had the time to propose possible redesigns of the system. Taking the time constraints into account, we obtained very satisfying results: the (often initially sceptical) developers were surprised to learn some unknown aspects of their system. On the other hand they typically knew already about many of the problems we found.

Industrial Curiosities. Due to non-disclosure agreements we are not allowed to report on the experiences we made with industrial case studies. Nonetheless, we have anonymized two screenshots taken from

Case Study	**Language**	**Size**		**Time Frame**
		Lines	Classes	
1	C++	1.2 MLOC	>2300 classes	1 Week
2	C++/Java	120 kLOC	>400 classes	1 Week
3	Smalltalk	600 kLOC	>2500 classes	3 Days
4	COBOL	40 kLOC	-	3 Days
5	C++	28kLOC	70 classes	2 Days
6	Smalltalk	-	700 classes	3 Days

Table 4.2: A list of some of the industrial case studies to which CodeCrawler was applied to.

the 1.2 MLOC C++ case study and show them below.

In Figure 4.14 we see a SYSTEM HOTSPOTS view of 1.2 million lines of code of C++. Although heavily compressed this view allows us to identify the largest classes in the system as well detect several hundred structs, displayed as small nodes at the top.

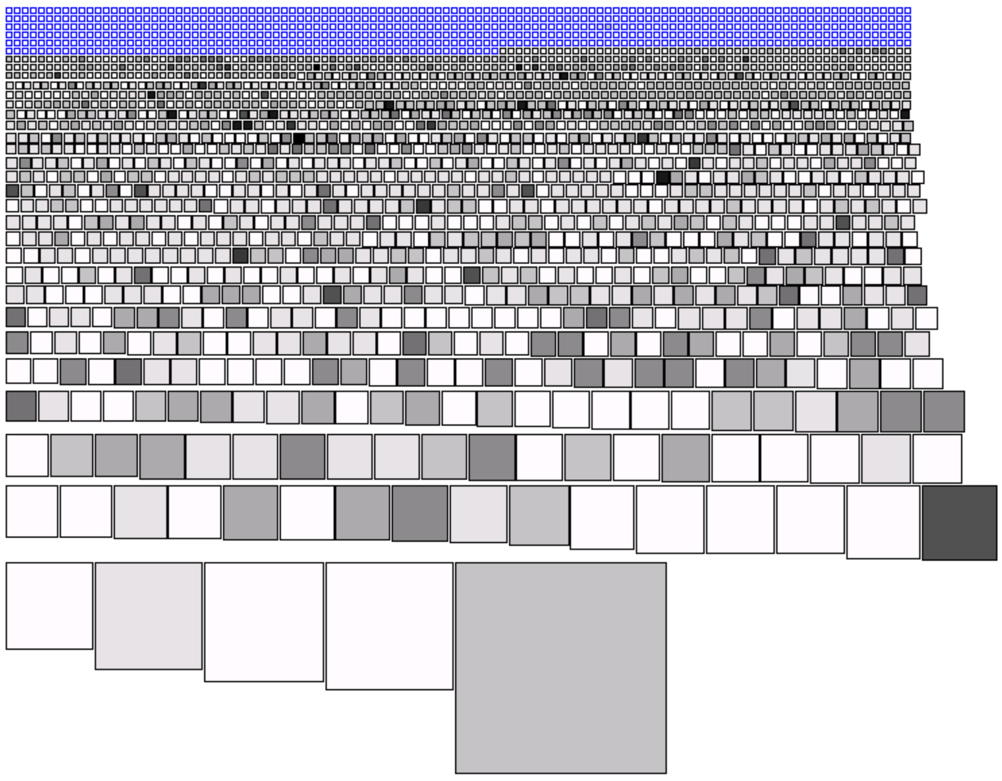


Figure 4.14: A SYSTEM HOTSPOTS view of a large industrial case study consisting of 1.2 million lines of code of C++. The nodes at the top represent classes without methods and are actually C++ structs.

In Figure 4.15 we see a SYSTEM COMPLEXITY view of one large hierarchy. The large amount of direct subclasses of the root class gives the hierarchy a particular shape: we called this pattern a *flying saucer* hierarchy.

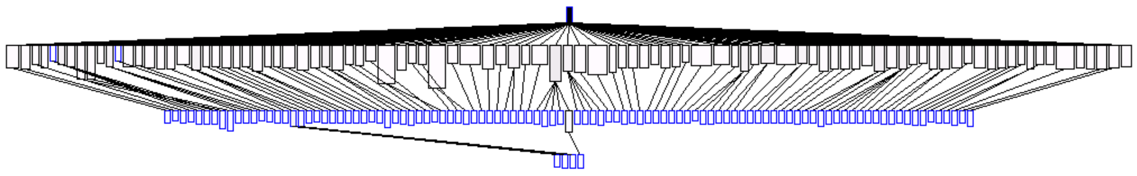


Figure 4.15: A SYSTEM COMPLEXITY view of a large hierarchy yields a form which we called *flying saucer*: the root class has 97 direct subclasses.

4.4 Related Work

Software Visualization

The graphical representations of software used in the field of software visualization, a sub-area of information visualization [WARE 00][Car 99], have long been accepted as comprehension aids to support reverse engineering. Indeed, software visualization itself has become one of the major approaches in reverse engineering. Price *et al.* have presented an extensive taxonomy of software visualization, with several examples and tools [PRIC 93].

Many tools make use of static information to visualize software, like Rigi [MÜ 86], Hy+ [CONS 93], SeeSoft [EICK 92], ShrimpViews [STOR 95], GSee [FAVR 01], and the FIELD environment [REIS 90], to name but a few prominent examples.

Substantial research has also been conducted on runtime information visualization, called program visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [LANG 95], Jinsight and its ancestors [PAUW 93][PAUW 99] and Graphtrace [KLEY 88]. Various approaches have been discussed like in [KAZM 95] or [JERD 97] where interactions in program executions are being visualized. In our current approach we do not exploit dynamic information. Richner has conducted research on the combination of static and dynamic information [RICH 02][RICH 99], where the static information is provided by the Moose Reengineering Environment.

Several systems make use of the third dimension by rendering software in 3D. Brown and Najork explore three distinct uses of 3D [Sta 98], namely (1) expressing fundamental information about structures that are inherently two-dimensional, (2) uniting multiple views of an object, and (3) capturing a history of a two-dimensional view. They exemplify these uses by showing screen dumps of views developed with the Zeus algorithm animation system [BROW 91]. However, they also state that “the potential use of 3D graphics for program visualization is significant and mostly unexplored”. Some of the systems cited above make both use of 2D and 3D visualizations.

Until now we have refrained from using 3D for our visualizations, mainly because it would contradict the lightweight constraint. However, we consider the exploration of the use of 3D as possible future work.

Metrics

Metrics have long been studied as a way to assess the quality and complexity of software [FENT 96], and recently this has been applied to object-oriented software as well [LORE 94] [HEND 96]. Metrics profit from their scalability and, in the case of simple ones, from their reliable definition. However, simple measurements are hardly enough to sufficiently and reliably assess software quality [DEME 99a]. Some metric tools visualize information using diagrams for statistical analysis, like histograms and Kiviat diagrams. Datrix [MAYR 96], TAC++ [FIOR 98a] [FIOR 98b] and Crocodile [LEWE 98] are tools that exhibit such visualization features. However, in all these tools the visualizations are mere side effects of having to analyze large quantities of numbers. In our case, the visualization is an inherent part of the approach, hence we do not visualize numbers, but constructs as they occur in source code.

Methodological Approach

To the best of our knowledge none of the approaches we reference in this thesis presents a reverse engineering approach, which can help a reverse engineer to apply a certain tool or technique. Storey *et al.* present in [STOR 99] some basic ideas on how to build a mental model during software exploration, but do not provide the much-needed, yet difficult to obtain, empirical evidence. We suppose this is because of the *ad hoc* nature of reverse engineering tools (including ours) and because software industry has not yet adopted such tools as concrete aids for their development process.

4.5 Conclusion

Software reverse engineering is a complex and difficult task, mainly because of the sheer size and complexity of software legacy systems. Several approaches have been developed to support the reverse engineering process, yet many of them fail because their scalability is limited or because they are too complex themselves. Two promising approaches, *software visualization* and *software metrics* both have their respective advantages and drawbacks. Our solution combines both approaches and exploits their advantages, while it minimizes their drawbacks.

4.5.1 Summary

In this chapter we have presented a reverse engineering approach based on clusters of *polymetric views*, lightweight visualizations enriched with metrics. This approach enables us to quickly gain insights into the inner structure of large software legacy systems and helps us to detect problems.

Furthermore, we have illustrated our approach by applying different views and by thus reverse engineering a case study. We have been able to understand different aspects of the case study, among which an overview of the application, a discussion on the used inheritance mechanisms, the detection of design patterns, the detection of several places where in-depth examinations are needed, as well as propositions on where possible refactorings could be applied.

The views presented in this chapter have been applied on several large industrial legacy systems written in different languages like Smalltalk, Java, and C++. The systems in question ranged in size from 100 kLOC to more than 1 MLOC. We are not allowed to report on the experiences obtained there due to non-disclosure agreements.

4.5.2 Benefits

The main benefits of our approach are the following:

- *Scalability.* Each polymetric view is able to transmit a great amount of information to the viewer in a condensed way. Furthermore, most of the presented views scale up to large systems, *e.g.*, more than 100 kLOC.
- *Simplicity*. The presented polymetric views are lightweight software visualizations, whose simplicity makes them easily adaptable to new contexts and programming languages.
- *Approach.* The presented approach provides guidance to a reverse engineer in the most delicate phase of a reverse engineering process, namely the beginning. By discussing the views in detail, the purpose and usability of each view is presented. We thus minimize the risk of getting lost during the reverse engineering.
- *Customizability.* The polymetric views are easily customizable, mainly by changing the metrics or the layouts. These changes are necessary, because every legacy system has particularities to which the polymetric views must be adapted to. In that sense the polymetric views are helpful because they emphasize the relative differences between the visualized entities. By that we mean that for example overly large classes will stand out in the SYSTEM HOTSPOTS view, unregarded from the average size of the classes in the subject system.

4.5.3 Limits

Our approach is limited in the following ways:

- *Visual language.* The presented views constitute a visual language which first must be learned by the viewer. In order to correctly interpret what he sees, the viewer must first learn what to look for in a polymetric view. Based on the amount of knowledge he gathers, the viewer can apply the approach more efficiently.
- *Ad-hoc approach.* The approach presented here cannot be used like a step-by-step process, but heavily depends on the context that a reverse engineer encounters. For example while in one case studies having methods of more than 100 lines may be exceptional, in another case it may be the average length. It does not mean that the latter system is worse in quality, it is just different.

4.5.4 Future Work

In the future we plan to investigate the following ideas:

- *Language specific views.* Since FAMIX is language-independent we have focused on developing views in this context. We believe there are views which exploit language specific information, for example modifier information in languages like C++ and Java or metaclasses in Smalltalk.
- *New entities and relationships.* The introduction of new entities and relationships, which may but do not need to have an equivalent in software could help to generate new views based on these new artifacts. A way to interactively generate these artifacts can be supported by grouping mechanisms, similar to the ones implemented in Rigi [MÜ 86], which group entities and relationships according to certain rules (i.e., naming conventions, types, etc.).
- *Usability and navigation.* The extensive use of direct-manipulation idioms [COOP 95], especially those relevant to the reverse engineering process , should further increase the malleability and flexibility of our tools. The introduction of navigation mechanisms, which reduce the latency between one view and the next one, can further increase the efficiency of the reverse engineering process.
- *3D.* The use of the third dimension can help exploit and visualize more semantic information, although we believe that this research generates results which cannot be classified as “lightweight” anymore.

Chapter 5

Fine-grained Software Visualization: The Class Blueprint

5.1 Introduction

It has been measured that in the maintenance phase software professionals spend at least half of their time analyzing software to understand it [CORB 89] and that code reading is a viable verification and testing strategy [BASI 87] [BASI 97] [HEND 02]. As we already mentioned, Sommerville [SOMM 00] and Davis [DAVI 95] estimate that the cost of software maintenance accounts for 50% to 75% of the overall cost of a software system. These facts show that understanding source code is one of the hardest tasks in the maintenance of software systems.

Furthermore nowadays *legacy systems* are not only limited to procedural languages but are also written in object-oriented languages. Contrary to what one may think, the object-oriented programming paradigm has but exacerbated this problem, since in object-oriented systems the domain model of the application is distributed across the whole system and the behavior is distributed across inheritance hierarchies with late-binding [WILD 92] [CASA 97] [Duc 99]. Moreover, reading object-oriented code is more difficult than reading procedural code [DEKE 02]. Indeed, in addition to the difficulties introduced by the technical aspects of object-oriented languages such as inheritance and polymorphism [WILD 92], the reading order of a class' source code is not relevant as it was in most of the procedural languages where the order of the procedures was important and the use of forward declarations required. This lack of reading order is emphasized in languages such as Smalltalk, *e.g.,* a language based upon a powerful integrated development environment (IDE) in which the concept of files is only used for external code storage but not for code editing. Moreover, even for file-based languages like Java, IDEs like Eclipse are literally eclipsing the importance of source files and putting forward *code browsing* practice as in Smalltalk.

In such a context understanding classes is of key importance as classes are the cornerstone of the object-oriented programming paradigm and the primary abstraction from which applications are built. Therefore there is a definitive need to support the understanding of classes and their internal structure. In the past, work has been done to support the understanding of object-oriented applications [KLEY 88] [LANG 95] [MEND 95]. Some other work focused on analyzing the impact of graphical notation to support program understanding based on control-flow [HEND 02]. However such approaches were on one hand powerful for supporting the identification of design patterns but on the other hand too generic and not fine tuned for the specific purpose of class understanding.

Summary. In this chapter we present a simple approach to ease the understanding of classes by visualizing a semantically augmented call- and access-graph of the methods and attributes of classes [1]. We only

[1]This chapter is an extended version of the article *The Class Blueprint: Visually Supporting the Understanding of Classes*, submitted for publication to IEEE Transactions on Software Engineering, which is based on our article *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*, published in the OOPSLA 2001 Proceedings (Conference on Object-Oriented Programming, Systems, Languages, and Applications), pp. 300 - 311, ACM Press, 2001.

take into account the internal static structure of a class and focus on the way methods call each other and access attributes and the way the classes use inheritance. We leave out the run-time behavior of a system.

We have coined the term *class blueprint*, a visualization of a semantically augmented call-graph and its specific semantics-based layout. The objective of our visualization is to help a programmer to develop a mental shape of the classes he browses and to offer a support for reconstructing the logical flow of method calls. In such a setup our approach reveals the “taste” of a class in terms of its call-graph and internal structure. However our approach does not magically provide a detailed understanding of a class’ functionality. Besides the presentation of the technical aspects that the class blueprint implies, we establish a vocabulary that we developed based on the insights we obtained during several case studies. This vocabulary identifies the most common and specific *visual patterns*, *i.e.*, recurrent graphical situations we encountered during the validation of this work over several large case studies. We believe that this vocabulary can be the basis of a language (in a similar vein to the use of design patterns [GAMM 95]) that reverse engineers can use when communicating with each other. We would like to stress the fact that the results presented in this chapter (and the whole thesis) are language independent as we base our work on a language independent meta-model for object-oriented source code representation [DEME 01]. However, most of our experiences have been conducted on applications developed in Smalltalk, although we applied our approach on case studies written in C++ and Java as well.

Contributions. The contributions of this chapter are the following:

- The definition of the *class blueprint*, a polymetric view which helps to achieve fine-grained reverse engineering goals. It helps to understand the concrete implementation of classes and class hierarchies, and detect common patterns or coding styles. Using this view we can look for signs of inconsistencies like the use of accessors. The class blueprint can be used to identify the possible presence of design patterns or occasions where design patterns could be introduced to ameliorate the system’s structure.
- The identification of *patterns* that represent recurring situations and the discussion of their meaning in terms of implementation.
- The definition of a vocabulary based on these patterns.

Structure of the chapter. We start by discussing the challenges that the understanding of classes involves, and by making precise the context of this chapter and its constraints (Section 5.2). In Section 5.3 we present the concept of the class blueprint and discuss two examples (Section 5.4). Based on the class blueprint, visual patterns are identified in Section 5.5. In Section 5.6 and Section 5.7 we then present several patterns and an analysis of the class blueprints (Section 5.8). We conclude (Section 5.11) the chapter with a discussion of the obtained results, the related work (Section 5.10), and an outlook on our future work.

5.2 The Challenge of Supporting Class Understanding

As our overall objective is to help reengineers to build a mental image of a class, we chose to visualize the essence of a class, therefore we restrict ourselves to methods, method invocations, attributes, and attribute accesses. According to the program cognition model vocabulary proposed by Littman *et al.* [LITT 96] we support an approach of understanding that is *opportunistic* in the sense that it is not based on a *systematic* line-by-line understanding but as needed. Moreover, to locate our approach in the general context of cognitive models [LITT 96] [VON 96], our approach is intended to support the *implementation plans* at the language level, *i.e.*, working at code chunks, here classes and methods.

Mayrhauser and Vans mention that the cognition processes work at all levels of abstraction simultaneously as programmers build a mental model of the code [VON 96]. Our approach is based on the visual identification of hotspots at the class level or hierarchy level which then are verified with opportunist code reading. In this sense, our claim is not that graphical visualization is better than text reading even if we believe that our approach eases the process [PETR 95]. Our approach creates a synergetic context between the two where blueprints reveal the way classes are built, help raising hypotheses, or questions that are then verified by reading some piece of code.

Classes are the cornerstone of object-oriented programming. They act as factories of objects and define the behavior of their instances. Understanding classes is then a primary task when reverse engineering an object-oriented legacy system. However classes are difficult to understand for the following reasons:

1. Contrary to procedural languages, the method definition order in a file is not important [DEKE 02]. There is no simple and apparent top-down call decomposition, even if some languages propose the visibility notion (private, protected, and public). This problem is emphasized in the context of integrated development environments (IDE) which disconnect the method definition from their physical storage medium. For example, in the Smalltalk environment, even if the IDE proposes advanced classifications such as method categories, the source file is just a storage medium and is not used to edit code.

2. Classes are organized into inheritance hierarchies in which at each level behavior can be extended, overridden, or simply added. Understanding how a subclass fits within its parent context is complex because late-binding provides a powerful instrument to build template and hook methods [GAMM 95] that allow children behavior to be called in the context of their ancestors. The presence of late-binding leads to "yoyo effects" when walking through a hierarchy and trying to follow the call-flow [WILD 92].

3. Classes define state and the procedures that act on this state. It is important to understand how the state is accessed, presented, if at all, to the class's clients, and how subclasses access this state.

Even if we display method invocations and attribute accesses we only consider the call-flow and not the control-flow of the methods. Furthermore, since classes do not stand alone, but exist within inheritance hierarchies, our approach supports the understanding of the class within an inheritance tree. Working at a call-flow level also supports the late-binding property of object-oriented programming in which a subclass can define methods that are called by superclass methods in replacement of their own methods.

For the visualization itself the solution we propose takes into account the physical limits of a screen, *i.e.*, a class blueprint must fit in one or exceptionally two screens of normal size. Bertin [BERT 74] assessed that one of the good practices in information visualization is to offer to the viewer visualizations which can be grasped at one glance (*e.g.*, without the need of scrolling or moving around). Furthermore the colors used in our visualizations also follow visual guidelines inferred by Bertin, Tufte [TUFT 90, TUFT 97], and Ware [WARE 00].

Work has already been done for supporting the understanding of object-oriented systems at the class level. GraphTrace proposes to visualize concurrent animated views to understand the way a system behaves [KLEY 88]. ObjectExplorer [LANG 95] uses both dynamic and static information that the reengineer can query and visualize via simple graphs to understand and verify his hypotheses. Using basic graph visualizations to represent various relationships, Mendelzon and Sametinger [MEND 95] show that they can express metrics, constraints verification, and design pattern identification. Cross *et al.*, more in the lines of procedural languages, have been proposing and validating new control structure diagrams to support the reading of the applications' control flow [CROS 98] [HEND 02].

The work presented in this chapter emerged from industrial code reverse engineering projects and is the result of several refinements to maximize the ease of understanding. Besides the industrial case studies on which we are not allowed to report, we performed several case studies on open-source software:

- Squeak, an open source multimedia Smalltalk which has been developed over the last years (1800 classes) [INGA 97].

- Duploc, a tool identifying code duplication in an language independent manner (160 classes).

- Moose, our own tool (200 classes).

Case study. In this chapter we use as case study the JUN framework: Jun is a freely available 3D graphic multi-media library that supports topology and geometry. It thus represents a fairly big system, which allows to have reproduceable and verifiable results. Its considerable size makes it a representative system for a reverse engineering. We analyzed version 398, which consists of more than 700 classes, 15’000 methods, and 2’000 attributes. The interesting aspect of Jun is its variety as it models a wide spectrum of domains: different format readers and writers, different composite structures (HTML, VRML), various complex rendering algorithms, even a Prolog interpreter and a Lisp compiler and interpreter. Jun is a mature and professionally developed system.

5.3 The Class Blueprint

This section introduces the concept of the *class blueprint*, a visual way of supporting the understanding of classes. A class blueprint is a semantically augmented visualization of the internal structure of a class, which displays an enriched call-graph with a semantics-based layout. It is augmented in various aspects that are explained in the subsequent sections:

- A class blueprint is structured according to *layers* that group the methods and attributes.
- The nodes representing the methods and attributes contained in a class are colored according to semantic information, *i.e.,* whether the methods are abstract, overriding other methods, returning constant values, etc.
- The nodes vary in size depending on source code metrics information.

5.3.1 The Layered Structure of a Class Blueprint

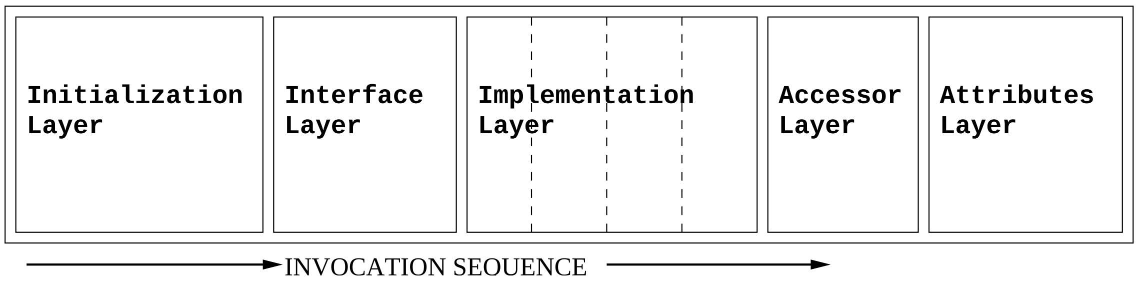


Figure 5.1: A class blueprint decomposes a class into layers.

A class blueprint decomposes a class into layers and assigns the attributes and methods of the class to each layer based on the heuristics described below. In Figure 5.1 we see an empty template of a class blueprint.

The layers support a call-graph notion in the sense that a method node on the left connected with another node on the right is either invoking or accessing the node on the right that represents a method or an attribute. From left to right we identify the following layers: *initialization layer, external interface layer, internal implementation layer, accessor layer, and attribute layer*. The first three layers and the methods contained therein are placed from left to right according to the method invocation sequence, *i.e.,* if method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and connected with an edge.

For each layer we present the conditions that methods must fulfill in order to belong to a certain layer. Note that the conditions listed below follow a lightweight approach and are not to be considered as complete. However, we have seen that they are sufficient for our purposes.

A class blueprint contains the following layers:

1. **Initialization Layer.** The methods contained in this first layer are responsible for creating an object and initializing the values of the attributes of the object. A method belongs to this layer if one of the following conditions holds:
    - The method name contains the substring “initialize” or “init”.

- The method is a constructor.
- In the case of Smalltalk, where methods can be clustered in method protocols, if the methods are placed within protocols whose name contains the substring "initialize".

In this layer there should also be the static initializers for Java, however we do not take them into account, as they are not covered by our metamodel [DEME 01].

2. **External Interface Layer.** The methods contained in this layer represent the interface of a class to the outside world. A method belongs to this layer if one the following conditions holds:

   - It is invoked by methods of the initialization layer.
   - In languages like Java and C++ which support modifiers (*e.g., public, protected, private*) it is declared as *public* or *protected*.
   - It is not invoked by other methods within the same class, *e.g.,* it is a method invoked from *outside* of the class by methods of collaborator classes or subclasses. Should the method be invoked both inside and outside the class, it is placed within the implementation layer.

   We do not include accessor methods to this layer, but to a dedicated layer as we show later on. We consider the methods of this layer to be the *entry points* to the functionality provided by the class.

3. **Internal Implementation Layer.** The methods contained in this layer represent the core of a class and are not supposed to be visible to the outside world. A method belongs to this layer if one of the following conditions holds:

   - In languages like Java and C++ if it is declared as *private*.
   - The method is invoked by at least one method defined in the same class.

4. **Accessor Layer.** This layer is composed of accessor methods, *i.e.,* methods whose *sole* task is to get and set the values of attributes.

5. **Attribute Layer.** The attribute layer contains all attributes of the class. The attributes are connected to the methods in the other layers by means of *access relationships* that connect the methods with the attributes they access.

5.3.2 Representing Methods and Attributes

We represent methods and attributes using colored boxes (nodes) of various size and position them within the layers presented previously. We map metrics information on the size of the method and attribute nodes, and map semantic information on their colors.

Mapping Metrics Information on Size

The width and height of the nodes reflect metric measurements of the represented entities, as illustrated in Figure 5.2. This approach has first been developed in the context of the *polymetric views* (see Chapter 4).

The class blueprint view visualizes method nodes and attributes nodes.

- *Method nodes.* In the context of a class blueprint, the metrics used for the nodes representing the methods are lines of code for the height and the number of invocations for the width.

- *Attribute nodes.* The metrics used for the boxes representing the attributes are the number of direct accesses from methods within the class for the width and the number of direct accesses from outside of the class for the height. The choice of these measures allows one to identify how attributes are accessed.

In Figure 5.3 we see how we distinguish a caller from a callee: the caller has outgoing edges at the bottom, while the callee has in-going edges at the top. Furthermore, the blueprint layout algorithm places the callee to the right of a caller.

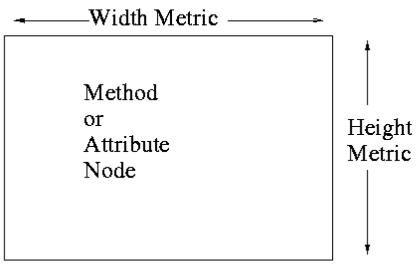


Figure 5.2: A graphical representation of methods and attributes using metrics: the metrics are mapped on the width and the height of a node.

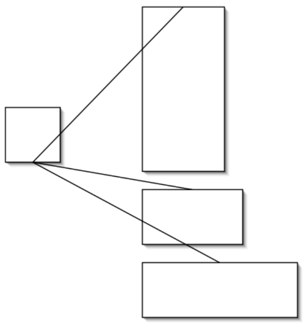


Figure 5.3: The caller has outgoing edges at the bottom, while the callee has in-going edges at the top.

Mapping Semantic Information on Color

The call-graph is augmented not only by the size of its nodes but also by their color. In a class blueprint the colors of nodes and edges represent semantic information extracted from the source code analysis. The colors play therefore an important role in conveying added information, as Bertin [BERT 74] and Tufte [TUFT 90] have extensively discussed. Table 5.1 presents the semantic information we add to a class blueprint and the associated colors.

Certain semantic information such as whether a method is delegating to another object is computed by analyzing the method abstract syntax tree (AST) and by identifying certain patterns. For example we qualify as delegating, a method invoking exactly the *same* method on an attribute (pattern 2) or a method invocation (pattern 1). In addition to those patterns we consider also the case when the method is returning a value using ˆ in Smalltalk (pattern 3 and 4). Note that such an analysis is language dependent but does not pose any problem in practice.

Pattern 1: delegating to invocation result.

```
methodX
   self yyy methodX
```

Pattern 2: delegating to an attribute.

```
methodX
   instVarY methodX
```

Pattern 3: delegating to an attribute with return.

```
methodX
   ˆ self yyy methodX
```

Pattern 4: delegating to invocation result with return.

Description	**Color**
Attribute	blue node
Abstract method	cyan node
Extending method. A method which performs a *super* invocation.	orange node
Overriding method. A method redefinition *without* hidden method invocation.	brown node
Delegating method. A method which delegates the invocation, *i.e.,* forwards the method call to another object.	yellow node
Constant method. A method which returns a *constant* value.	grey node
Interface and Implementation layer method.	white node
Accessor layer method. Getter.	red node
Accessor layer method. Setter.	orange node
Invocation of a method.	blue edge
Invocation of an accessor. Semantically it is the same as a direct access.	blue edge
Access to an attribute.	cyan edge

Table 5.1: In a class blueprint semantic information is mapped on the colors of the nodes and edges.

```
methodX
   ^  instVarY methodX
```

The fact that a method is abstract is also extracted from the analysis of the method AST as in Smalltalk the only way to specify that a method is abstract is to invoke the method `subclassResponsibility` (see Pattern 5). For Java and C++, specific language constructs make the analysis simpler.

Pattern 5: Abstract method.

```
methodX
   self subclassResponsibility
```

Note that the color associations shown in Table 5.1 are not mutually exclusive. Therefore, a node could have more than one color assigned to it. In such a case the color determined by the source code analysis takes precedence over the color given by the layer a certain node belongs to, as this information conveys usually more semantics.

5.3.3 The Layout Algorithm of a Class Blueprint

The algorithm used to layout the nodes in a class blueprint first assigns the nodes to their layers and then sequentially lays out the layers. Within each of the first three layers, nodes are placed using a horizontal tree layout algorithm: if method *m1* invokes method *m2*, *m2* is placed to the right of *m1* and both are connected by an edge which represents the invocation relationship. In case a method *m1* accesses an attribute *a1*, the edge connecting *m1* and *a1* represents an access relationship, as is denoted by the color of the edge. In the last two layers the nodes are placed using a vertical line layout, *i.e.,* the nodes are placed vertically below each other. Although the layout algorithm can be considered lightweight, it shows acceptable results in terms of visual quality. The complex structure of a method invocation graph allows for cycles because of recursive calls, therefore the tree layout algorithm used as part of the overall blueprint layout is cycle-resistant.

In Figure 5.4 we see a template blueprint. We see that there are 2 initialization methods and 3 interface methods. We also see that some of its accessors (the ones in the ellipse) are not invoked and therefore unused and that one of the attributes (A) is not accessed by the methods of this class. The next section presents two real class blueprints in detail.

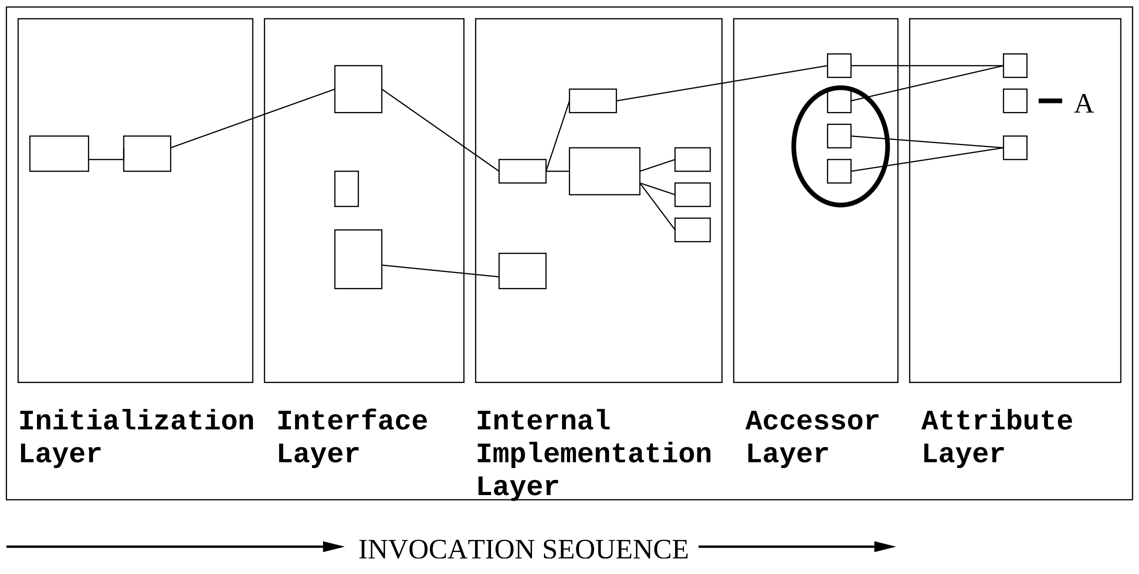


Figure 5.4: The methods and attributes are positioned according to the layer they have been assigned to.

5.4 Detailing Blueprints

To show how the class blueprint visualization allows one to represent a condensed view of a class' methods, call flow, and attribute accesses, we detail two classes which implement two different domain entities of the JUN framework: the first one defines the concept of a 3D graph for OpenGL mapping and the second is a rendering algorithm. We present the blueprints and some piece of code to show how the graphical representation is extracted from the source code and how the graphical representation reflects the code it represents, building a trustable model. To help the reader to understand the first blueprint we also show on the right of the figure a deviated blueprint in which the method names are shown on the boxes that represent them. The left part of Figure 5.5 shows the blueprint of the class `JunOpenGL3dGraphAbstract` which we describe hereafter. As the named blueprint on the right in Figure 5.5 shows, this kind of representation does not scale well in practice.

The code shown is Smalltalk code, however being fluent in Smalltalk is not important as we are only concerned by method invocations and attribute accesses[2].

Note that some of the figures may contain several patterns whose discussion not always precedes the figures. However, the captions of the figures make use of the complete pattern vocabulary presented in this chapter.

5.4.1 Example 1: An Abstract Class

The class blueprint shown in Figure 5.5 has the following structure:

- **One initialization layer method.** This method called `initialize` is positioned on the left. As shown, it extends (invokes) a superclass method with the same name, hence the node color is orange. It accesses directly two attributes as the cyan line shows it. The code of the method `initialize` is the following one:

```
initialize
```

[2]In Smalltalk, attributes as local variables are read simply by using the attribute name in an expression. They are written using the `:=` construct. In a first approximation messages follow the pattern `receiver methodName1: arg1 name2: arg2` which is equivalent to the C++ syntax receiver.methodName1name2(arg1, arg2). Hence `bidiagNorm := self bidiagonalize: superDiag` assigns in the variable `bidiagNorm` the result of the method `bidiagonalize`.

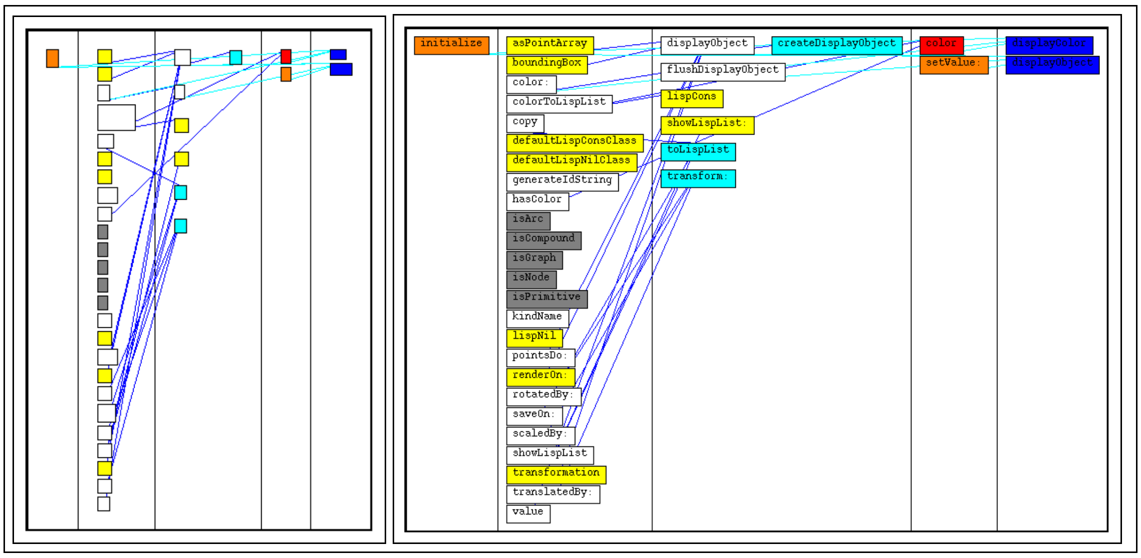


Figure 5.5: Left: An blueprint of the class *JunOpenGL3dGraphAbstract*, which represents OpenGL three-dimensional graphs. Right: The same class displayed with method names.

```
super initialize.
displayObject := nil.
displayColor := nil
```

- **Several external interface layer methods.** Note that many of them have a yellow color, *i.e.*, they delegate the functionality. The following method `asPointArray` is a delegating method.

  ```
  asPointArray
     ^self displayObject asPointArray
  ```

  The reader may be intrigued by the fact that there are yellow nodes, hence delegating methods, that neither invoke other methods nor access any attribute. This is the case of the `lispCons` method whose code is shown hereafter. In fact such methods delegate the invocation to the metaclass. This happens because of Smalltalk semantics that specify that any class is an instance of its metaclass. It is good practice to factor constants at the metaclass level as in the present case. A similar situation would occur in Java when a method delegates to a static variable. Note that we decided not to introduce a specific analysis to cope with this Smalltalk specific point to let our approach be as general as possible.

  ```
  lispCons
     ^self class lispCons
  ```

  The five grey nodes in the interface layer are methods returning constant values as illustrated by the following method `isArc`. This method illustrates a typical practice to share a default behavior among the hierarchy of classes.

  ```
  isArc
     ^ false
  ```

- **A small internal implementation layer with two sub-layers.** Here we show that the blueprint granularity resides at the method level, as the visualization does not specifically represent control flow constructs. The method `displayObject` performs a lazy initialization, *i.e.*, it initializes the attributes only when the attributes are accessed and acts as a template method [GAMM 95] by calling the method `createDisplayObject` which is represented as a cyan node as it is abstract. The method `createDisplayObject` should then be defined in the subclasses.

```
displayObject
   displayObject isNil
      ifTrue:
         [displayObject := self createDisplayObject].
   ^displayObject

createDisplayObject
   ^self subclassResponsibility
```

- **Two accessors.** There is a read-accessor, `color`, displayed as the red accessor node and a write-accessor, `setValue:` displayed as the rightmost orange accessor node.
- **Two attributes.** Note that the read-accessor reads one attribute, while the write-accessor writes the other one. However no method uses the write-accessor. The attributes are also directly accessed: the `initialize` method accesses both, while two other methods do also directly access the attributes which is an inconsistent coding practice.

5.4.2 Example 2: An Algorithm

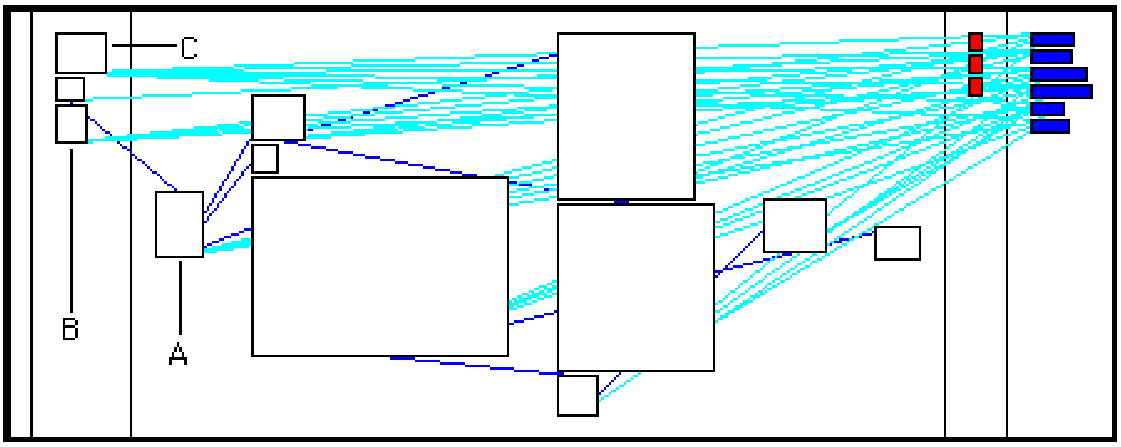


Figure 5.6: A blueprint of the class *JunSVD*. This class blueprint shows patterns of the type *Single Entry*, *Structured Flow* and *All State*.

The second class blueprint presented in Figure 5.6 displays the class `JunSVD` implementing the algorithm of the same name. Looking at the blueprint we get the following information.

- **No initialization layer method.** This is reflected by the fact that the left layer is empty.
- **Three external interface layer methods.** Two of them access directly the attributes of the class. We also see that the second external interface layer method is actually an entry point to all the methods in the internal implementation layer.
- **An internal implementation layer composed of nine methods in five sub-layers.** The class is actually written in a clearly structured way. Therefore the class blueprint can also be used to infer a reading order of the methods contained in this class. The blueprint shows us that for example, the node **A** which represents the method `compute` shown hereafter invokes the methods `bidiagonalize:`, `epsilon`, and `diagonalize:with:`.

```
compute
   | superDiag bidiagNorm eps |
   m := matrix rowSize.
   n := matrix columnSize.
   u := (matrix species unit: m) asDouble.
   v := (matrix species unit: n) asDouble.
   sig := Array new: n.
   superDiag := Array new: n.
   bidiagNorm := self bidiagonalize: superDiag.
   eps := self epsilon * bidiagNorm.
   self diagonalize: superDiag with: eps.
```

- **Three read accessor methods.** Although three read-accessors have been defined, they are not used by methods of this class, because they do not have any in-going blue edges that would exemplify their use.
- **Six attributes.** All the attributes in this class are accessed by several methods, *i.e.*, all the state of the class is accessed by the methods. The blueprint also reveals that the attributes are heavily accessed. The nodes marked as *A*, *B*, and *C* consistently access *all* the attributes `matrix`, `n`, `m`, `sig`, `v`, and `u`. To understand how this particular behavior is possible we show the code of the method `generalizedInverse` (*C*). After reading the code we understand easily that this particular behavior for a class is normal for an algorithm and we can mentally acknowledge that the other methods are built in a similar fashion.

```
generalizedInverse
   | sp |
   sp := matrix species new: n by: m.
   sp doIJ:
      [:each :i :j |
         sp
           row: i
           column: j
           put: ((i = j and:
                  [(sig at: j) isZero not])
                     ifTrue: [(sig at: j) reciprocal]
                     ifFalse: [0.0d])].
   ^(v product: sp)
       product: u transpose
```

5.4.3 Class Blueprints and Inheritance

Understanding classes in presence of inheritance is difficult as the flow of program is not local to a single class but distributed over hierarchies, as mentioned by Wild [WILD 92] and Lange [LANG 95]. Class blueprints increase their value when seen in the light of inheritance. In this case we visualize every class blueprint separately and put the subclasses below the superclasses according to a simple tree layout.

In Figure 5.7 we see a concrete inheritance hierarchy of class blueprints. The superclass defines some behavior that is then specialized by each of the three subclasses named `JunColorChoiceHSB`, `JunColorChoiceSBH`, `JunColorChoiceHBS`. The blueprint of this hierarchy reveals immediately that the subclasses have been developed to fit exactly the superclass and nothing more. The subclasses do not define any extra behavior, the superclass is the class to be analyzed in order to understand the whole hierarchy.

We see that the root class defines several abstract methods that represent color component such as brightness, hue, and color (denoted by the cyan color) and which are overridden (denoted by the brown color) in the three small subclasses. As there is the same number of brown nodes than cyan one, there is a good chance that the subclasses are concrete classes.

The method named `color` (*A*) is a template method [GAMM 95] that calls three abstract methods as confirmed by the definition of the method `color` hereafter.

```
color
  ^ColorValue
```

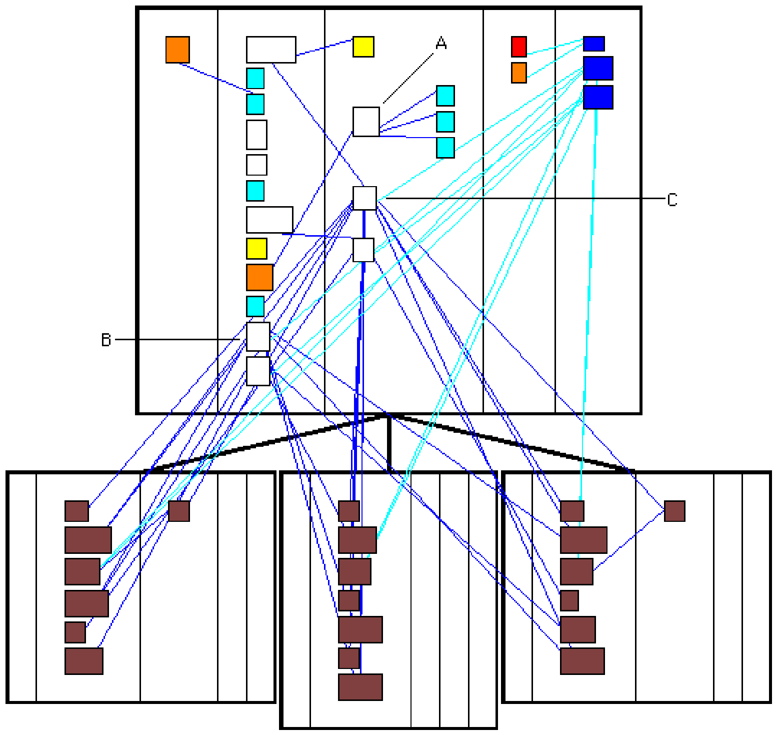


Figure 5.7: A class blueprint visualization of an inheritance hierarchy with the class *JunColor-Choice* as root class. The root class contains an *Interface* pattern, while each of the subclasses is a pure *Overrider*. Furthermore, each subclass is a pure *Siamese Twin*.

```
hue: self **hue**
saturation: self **saturation**
brightness: self **brightness**
```

We see that the methods `xy:` (*B*), and `xy` (*C*) play a central role in the design of the class as they are both called by several of the methods of each subclass, as confirmed by the following method of the class `JunColorChoiceSBH`:

```
JunColorChoiceSBH>>brightness: value
   ((value isKindOf: Number)
      and: [0.0 <= value and: [value <= 1.0]])
         ifTrue: [self xy: self xy x @ 1 - value]
```

These examples show that the blueprints are useful to fulfill our fine-grained reverse engineering goals, namely:

1. Understand the concrete implementation of classes and class hierarchies, and detect common patterns or coding styles. Look for signs of inconsistencies like the use of accessors.
2. Identify the possible presence of design patterns or occasions where design patterns could be introduced to ameliorate the system's structure.
3. Build a mental image of a class in terms of method invocations and state access.
4. Understand the class/subclass roles.
5. Identify key methods in a class.

Blueprints act as revealers in the sense that they raise questions, support hypotheses or clearly show important information. When questions are raised, code reading helps confirming the visualization hints or information. However, code reading is not always necessary but used sparingly on identified methods. There is a definitive synergy between the visual images generated by the blueprint and the code reading. Class blueprints allow one to characterize classes but also represent an important communication means, as we present in the coming sections.

5.5 A Vocabulary based on Patterns in the Class Blueprints

While the approach is already an excellent vehicle to support the understanding of classes, it also provides the basis to develop a visual vocabulary that enables programmers to communicate recurrent situations they encounter. Indeed, recurrent situations in the code produce similar *blueprint patterns* in terms of node colors and flow structure. These (blueprint) patterns stem from the experiences we obtained while applying our approach on several industrial case studies. We subdivide the discussion of the patterns in two separate sections depending on the context in which a blueprint is presented:

1. **Single class perspective**, where we look at a single blueprint without considering surrounding sub- or superclasses (Section 5.6).

2. **Inheritance perspective**, where we extend the context to the inheritance hierarchy where the class resides (Section 5.7).

We use the term *pure* class blueprint when it is composed of only one and exclusively one pattern. Note that the only kind of collaboration between classes we discuss in this chapter is inheritance.

It is important to understand that even if some of the patterns could be automatically identified by our tool, the identification of patterns is based on a human interpretation of a blueprint. There are advantages and disadvantages letting the reengineer identify patterns: the advantages are that the human mind can deal with non-regular information and still extract useful pieces, which is really important in the current context. The disadvantage are that the reengineer should be trained to analyze the blueprint and that he may wrongly interpret the pattern. However, this is normally not a problem as the code mapping is simple and the reengineer can quickly look at the code to confirm his hypothesis. In the future we want to evaluate how to automate the identification of non-regular and trivial patterns and whether this is worth the effort.

5.6 Single Class Blueprint Patterns

In this part we present the patterns that blueprints contain without considering surrounding sub- and superclasses. Note that one class blueprint may contain several patterns. The blueprint patterns in this section are grouped according to the following criteria:

- *size* (Section 5.6.1)
- *distribution layer* (Section 5.6.2)
- *semantics* (Section 5.6.3)
- *call-flow* (Section 5.6.4)
- and *state usage* (Section 5.6.5)

Note that this grouping is not strict and is mainly used to ease the reading of the chapter.

5.6.1 Size-based Blueprint Patterns

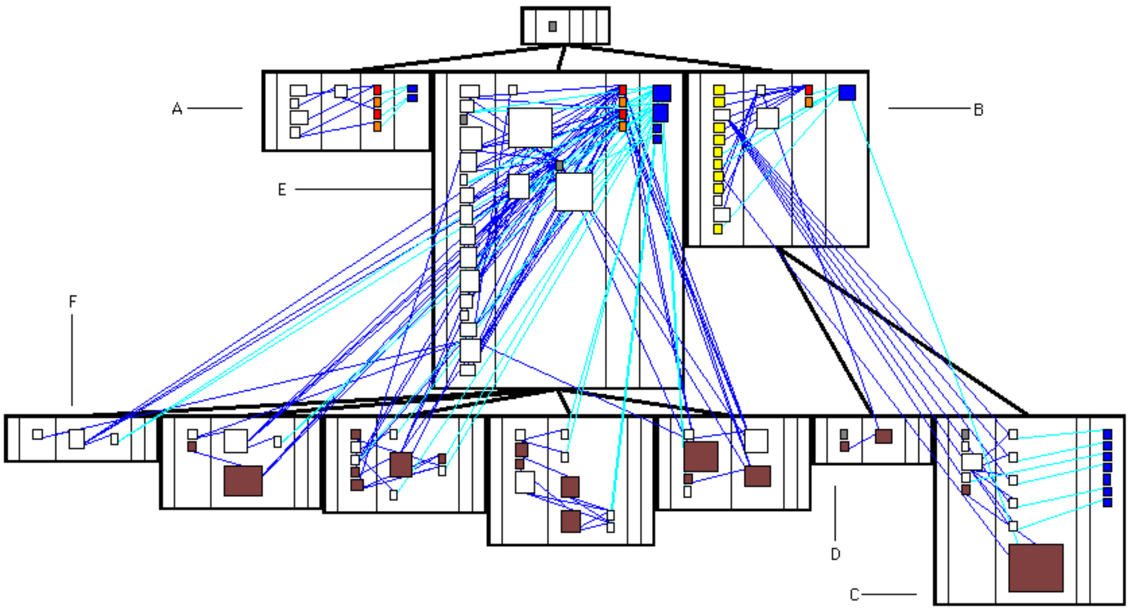


Figure 5.8: A class blueprint visualization of an inheritance hierarchy with the class *JunPrologEntity* as root class.

Four simple patterns describe classes regarding their size: *Single*, *Micro*, *Large Implementation*, and *Giant*.

Single

This pattern is composed of one node. It describes classes that only consist of one method (see the root class of the hierarchy in Figure 5.8). This happens in the following cases:

1. The class in question represents dead code or has not been completely implemented yet.
2. It is the result of code sharing among hierarchies. It often represents methods defining single default values or testing methods in the form of *isSomething()* as shown by the following method definition. See the discussion of the *Single Constant Definer* pattern for more details. The single method of the root class in Figure 5.8 has the following definition:

```
JunPrologEntity>>isJunPrologEntity
   ˆtrue
```

When the method is not a method simply defining a constant but has a certain complexity, it is worth to look at it as it represents common behavior shared among several classes and often used to distinguish between several kinds of classes. In an inheritance hierarchy consisting of fifteen classes, not shown as blueprints in this chapter, implementing probability distribution, we found two single classes at the top of the hierarchy whose main purposes was to factor out two different ways of computing a distribution as shown by the following definitions:

```
JunDiscreteProbability>>distribution: aCollection
   | t |
   t := 0.0.
   aCollection do: [:i | t := t + (self density: i)].
   ˆt
```

```
JunContinousProbability>>distribution: aCollection
   | t aStream x1 x2 y1 y2 |
   t := 0.0.
   aStream := ReadStream on: aCollection.
   x2 := aStream next.
   y2 := self density: x2.
   [x1 := x2.
   x2 := aStream next]
       whileTrue:
         [y1 := y2.
         y2 := self density: x2.
         t := t + (x2 - x1 * (y2 + y1))].
    ˆt * 0.5
```

3. This may occur when classes are subclasses of large classes of which they specialize only a limited default behavior or constant definitions.

Large Implementation

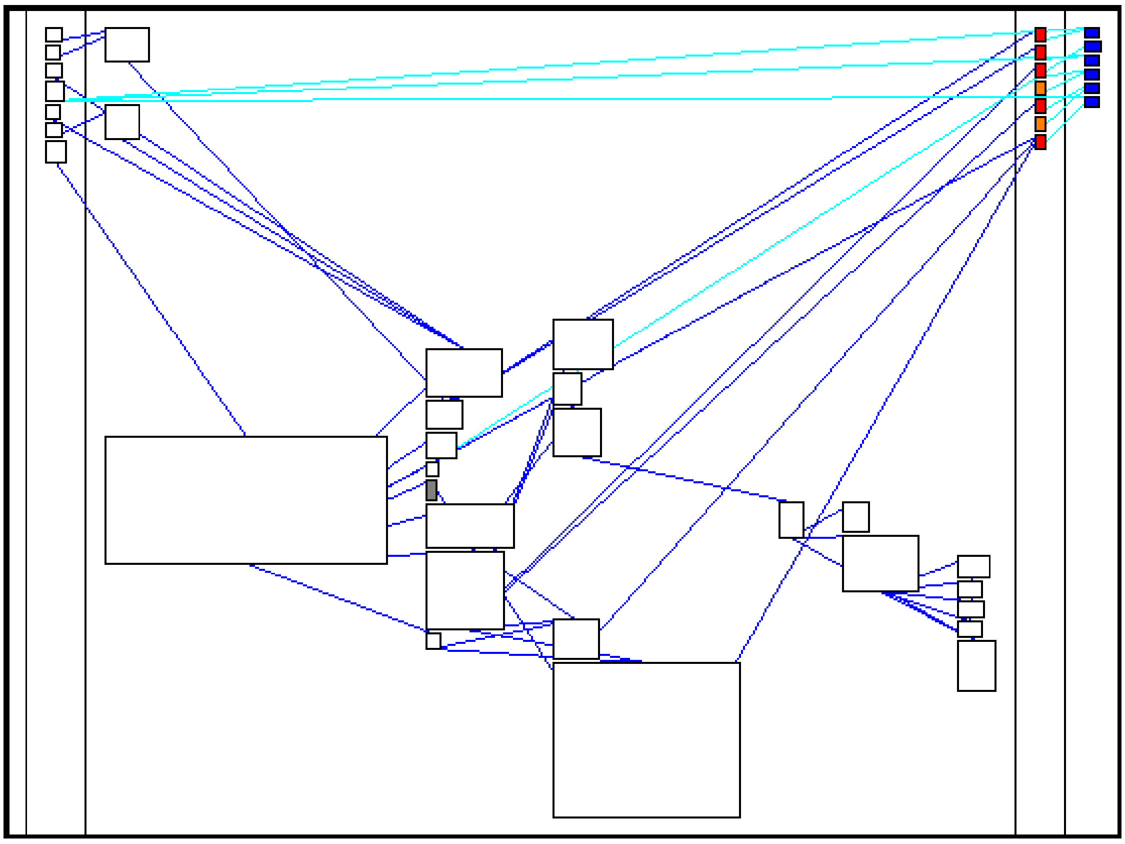


Figure 5.9: The blueprint of the class *JunSourceCodeSaver* contains a *Large Implementation*, a *Single Entry*, and a *Structured Flow* pattern at the bottom.

This pattern is characterized by implementation layers containing many nodes often structured in several sub-layers. The overall percentage in nodes number and screen space of the implementation layer dominates all the other layers. It describes classes that have a large implementation decomposed in several methods with numerous invocations between those methods. In Figure 5.9 we see that the class `JunSourceCodeSaver` has a small public interface and a large internal implementation layer with large methods and 6 sub-layers. The role of this class is to save in a proprietary format the code of the application. As in Smalltalk classes are objects too, the code of this class is mainly extracting information from classes and transforming into strings that are finally stored on disk. This explains the impressive size of certain methods.

Micro

This pattern is composed of only a couple of nodes. It describes a small class that is composed of only a couple of methods (see class annotated as emphF in Figure 5.8). This often occurs in subclasses that specialize behavior.

Giant

A really large number of nodes and invocations composes the entire blueprint. This pattern describes a huge class that is composed of hundreds of methods. Normally the blueprint layout algorithm is not efficient enough to support the understanding of such classes, although often patterns are recognizable. Such classes can have a complex initialization structure producing very long methods. Usually classes revealing a *Giant* pattern are classes having too much responsibilities and thus require a closer inspection.

5.6.2 Layer Distribution-based Blueprint Patterns

Three patterns *Three Layers*, *Wide Interface*, and *Interface* are based on the distribution of methods in the blueprint layers.

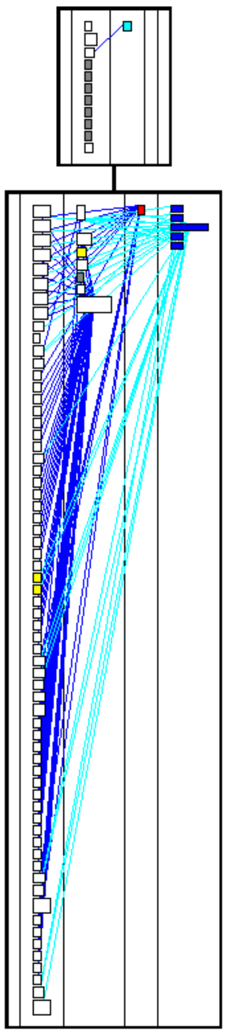


Figure 5.10: The root class is a combination of a nearly pure *Interface* and a *Constant Definer* pattern, while the subclass (*JunAngle*) is a combination of a *Wide Interface* and a *Funnel* pattern.

Three Layers

Graphically this pattern is composed of three to four colored bands with few nodes: one or two white bands for the interface layer, one red for the accessor, and one blue for attributes. This pattern describes classes that have few methods, some accessors, and some attributes. Usually these classes are small and implement primitive behavior and access to data. In Figure 5.8 we see that the class annotated as *A* belongs to this category.

Interface

Graphically the pattern present one predominant interface layer. It occurs when a class acts as an interface, which is frequent for abstract superclasses. It also occurs when the class acts as a pool of constants. In the Smalltalk programming language there is no construct for defining constant values, therefore class methods are often used to return constant values. Such classes can also contain a *Constant Definer* pattern as shown by the top class blueprint in Figure 5.10.

Wide Interface

Graphically this pattern is composed of a large interface layer proportionally to the rest of the class. A *Wide Interface* blueprint is one that offers many entry points to its functionality proportionally to its implementation layer (see Figure 5.10 and to a certain extent Figure 5.5). Examples of such classes are GUI classes with many buttons on the user interface which implement a method for every button the user can press.

5.6.3 Semantics-based Blueprint Patterns

As in class blueprints we map semantic information to node and edge colors, we identify the patterns *Delegate*, *Data Storage*, *Constant Definer*, *Accessor User*, *Direct Access*, and *Access Mixture* by looking at which colors are present in a blueprint and where the nodes with those colors are located.

Delegate

Graphically this pattern is composed of yellow nodes often found in the interface layer. *Delegate* describes a class which defines delegating methods, *i.e.*, it forwards invocations to attributes or to accessor invocations. A *Delegate* can be an indication for design patterns such as *Facade* or *Wrapper* [GAMM 95]. The class in Figure 5.11 (repeated from Figure 5.5) shows a *Delegate* pattern. The class annotated as *B* in Figure 5.8 also presents a *Delegate* pattern.

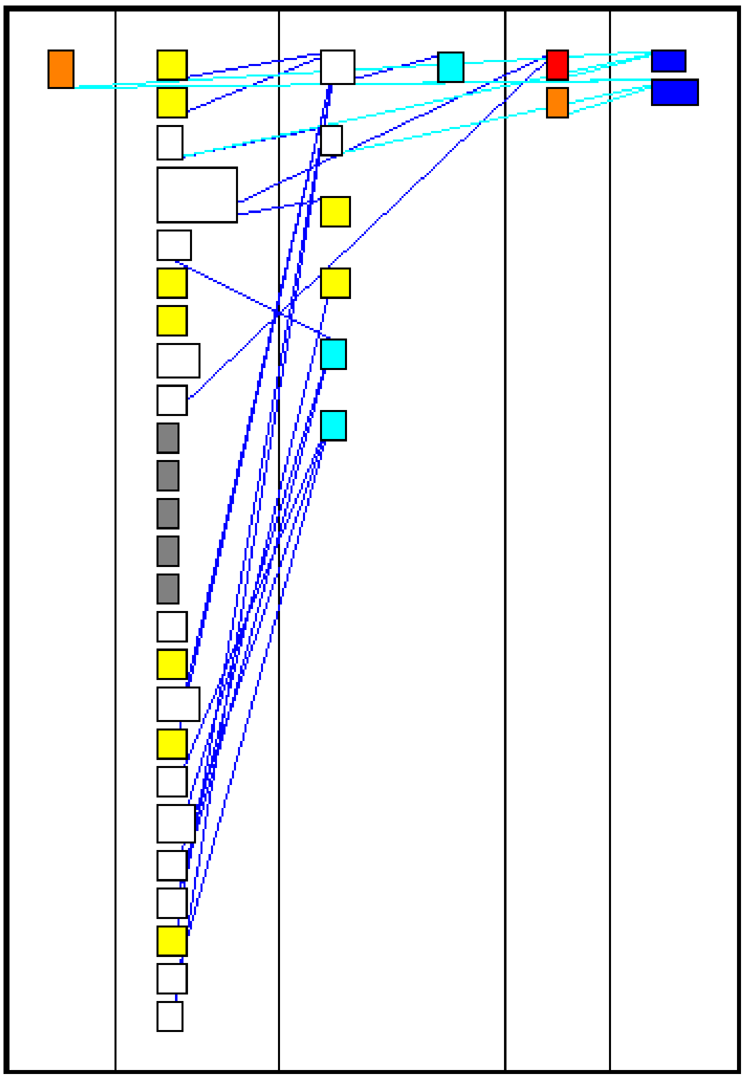


Figure 5.11: The class blueprint of the class *JunOpenGL3dGraphAbstract*: it mainly consists of a *Delegate* pattern.

Data Storage

Graphically this pattern presents mainly two layers, one red of accessors and one blue of attributes, and sometimes also has one extra method to initialize the attributes. The *Data Storage* pattern describes a class which mainly defines accessors to attributes. Such a class usually does not implement any complex behavior, but merely stores and retrieves data for other classes. The implementation layer is often empty. Looking for duplicated logic in the clients of such classes is usually a good way to reduce duplicated code and to enforce law of Demeter [LIEB 89], [DEME 02].

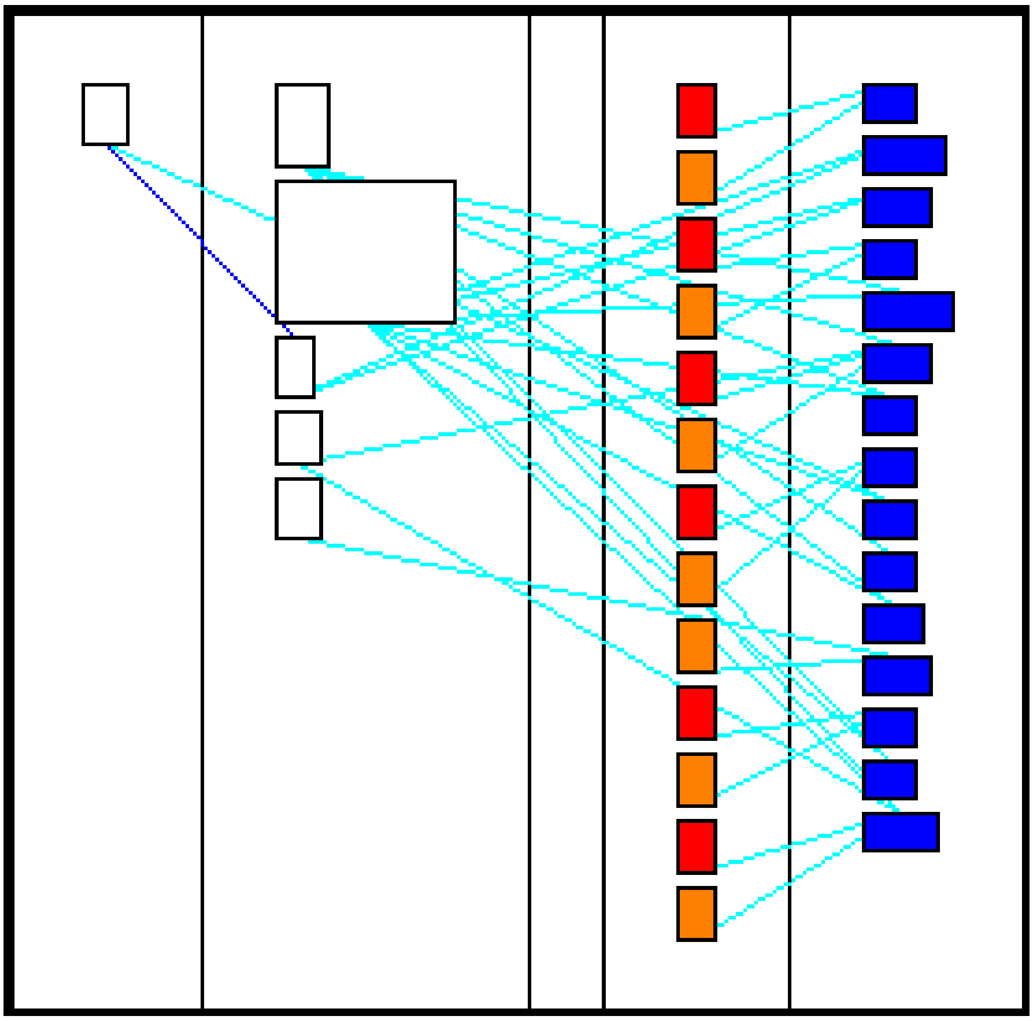


Figure 5.12: The class blueprint of the class *JunJfifColorComponent*: it contains a *Data Storage* and a *Three Layers* pattern.

Figure 5.12 shows a class presenting some aspects of the *Data Storage* pattern, but is not limited to this. This class could also be categorized as a *Three Layers* even if the number of accessors is large compared to the other methods defined in the class. Not being able to exactly categorize the class is not a problem as the key point is that the reengineer now knows that the class seems to act as a data repository with some extra behavior. Reading briefly the method `nextSample` (the biggest method node in this blueprint), confirms this hypothesis as this method generates new colors using the attributes of the object.

Constant Definer

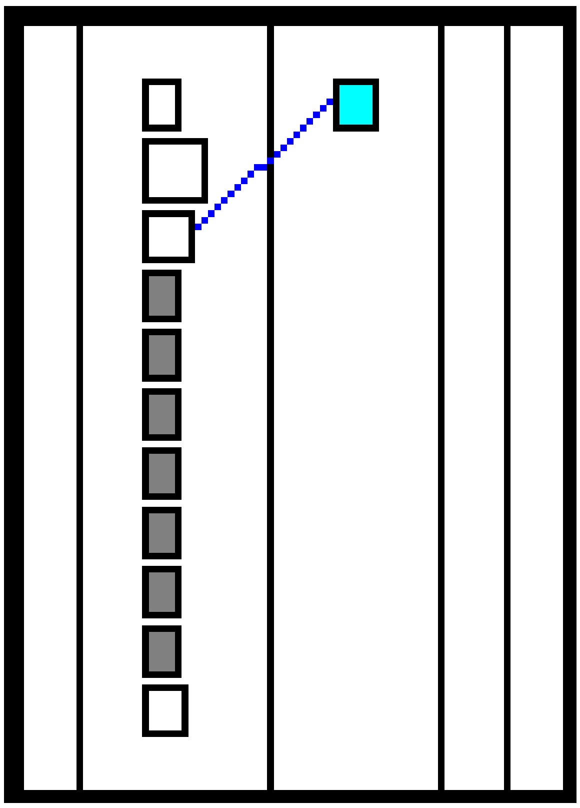


Figure 5.13: The class blueprint of the class *JunAngleAbstract*: it contains a distinct *Constant Definer* pattern.

Graphically this pattern is composed of grey nodes often residing in the interface layer. It describes a class which defines methods that return constant values such as integers, booleans, or strings. Pure *Constant Definer* blueprints are rare as a class is seldom limited to define constants. The class blueprint in Figure 5.13, which is the root class shown in Figure 5.10, and the one in Figure 5.5 both contain a *Constant Definer* pattern.

Accessor User* / *Direct Access* / *Access Mixture

Graphically these three patterns are linked to the consistency in which edges arrive to attributes and accessors. These three patterns which are mutually exclusive describe the use of accessors in classes.

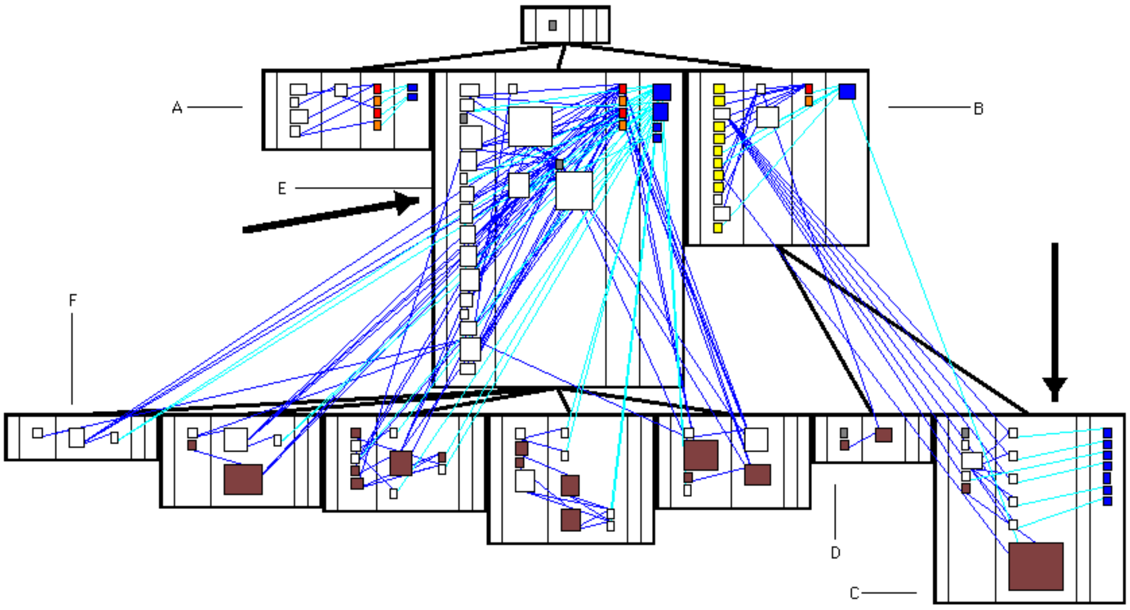


Figure 5.14: A class blueprint visualization of an inheritance hierarchy with the class *JunPrologEntity* as root class.

In the case of *Accessor User*, two accessors (the getter and the setter) have been consistently defined for every attribute in the class and the attributes are not accessed directly. In the case of *Direct Access*, no accessors at all have been defined, and the attributes are always accessed directly. In the case of *Access Mixture*, there is an inconsistent definition and use of the accessors. These patterns reveal the programming styles and whether they are followed. It is an important information when lazy initialization has to be introduced in the class as all the accesses to the state should be done via a single method implementing the lazy schema. In Figure 5.8 the class blueprint *A* shows an *Accessor User* pattern, *i.e.*, for every attribute there are two accessors and the attributes are only accessed via the accessors. In Figure 5.14 the class *C* is an example of a *Direct Access* pattern, while within the same hierarchy the class blueprint *E* shows a *Access Mixture* pattern. This indicates a lack of coding conventions.

5.6.4 Call-flow-based Blueprint Patterns

Based on the call-flow between the methods, we identify the following patterns: *Single Entry*, *Structured Flow*, *Method Clumps*, and *Funnel*.

Single Entry

Graphically this pattern is composed of a minimal, often limited to one node, interface layer but connected to all the nodes of the larger implementation layers. *Single Entry* describes a class which has very few or only one method in the external interface layer acting as entry point to the functionality of the class. It then has a large implementation layer with several levels of calls. Such classes are designed to deliver only little yet complex functionality. Classes which implement a specific algorithm (*e.g.*, parsers) show this pattern. Figure 5.15 shows two *Single Entry* patterns in one class blueprint. The two distinctive entry points are the root nodes of two separate method invocation trees. We deduce that the class provides for two separate, probably complementary, functionalities as they access the same attributes.

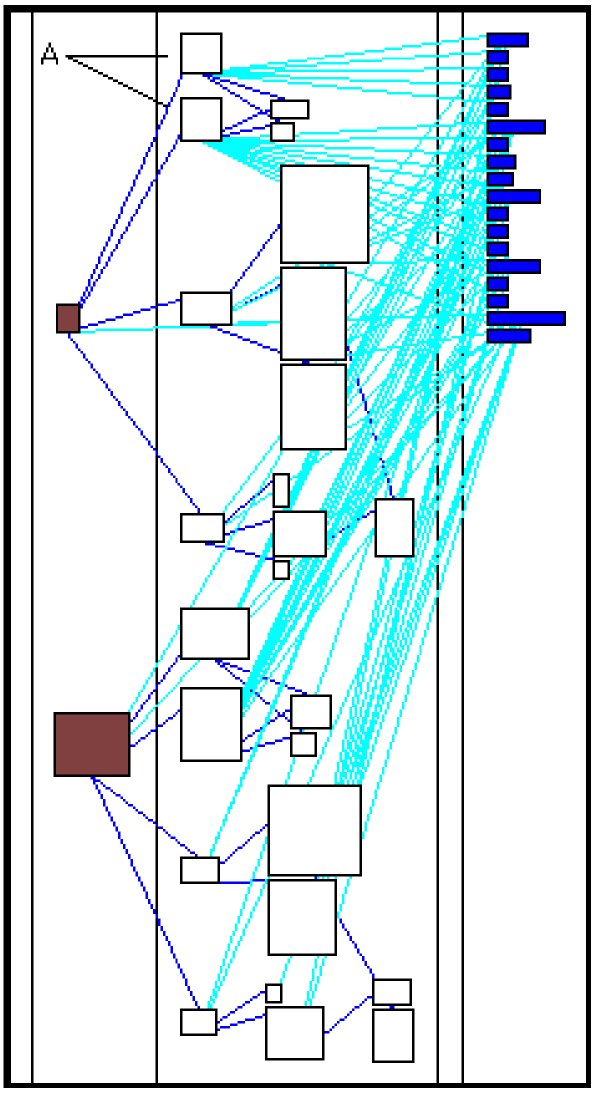


Figure 5.15: The class blueprint of the class *JunBmpImageStream* with two *Single Entry* patterns.

Structured Flow

Graphically this pattern presents a cluster of methods structured in a deep and often narrow invocation tree. This pattern reveals that the developer has decomposed an implementation into methods that invoke each other and possibly reuse some parts. It supports the reading of the methods. A typical example is the decomposition of a complex algorithm into pieces. The bottom of the class blueprint in Figure 5.16 shows a well pronounced *Structured Flow* pattern.

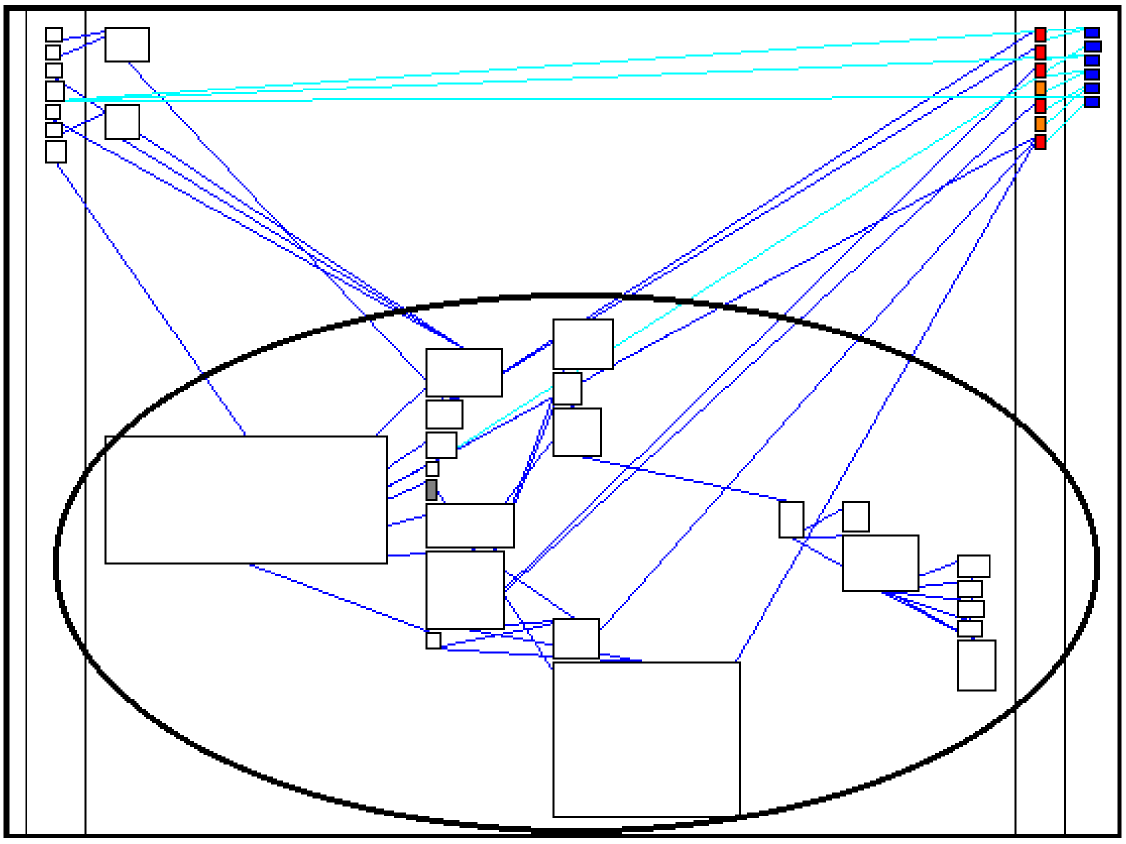


Figure 5.16: The *Structured Flow* pattern of the class *JunSourceCodeSaver*.

Method Clumps

Graphically this pattern is composed of one large or huge node surrounded by some tiny nodes. It contains clusters of methods each with one very large method that is calling many of small methods. The large methods are not structured following a functional decomposition, but have a monolithic structure (one big chunk of code). Figure 5.17 shows two *Method Clumps* patterns, the large nodes represent methods having more than 100 lines of code. They are the direct translation of the GNU diff algorithm written in C. To give an idea of the disproportionality between those methods and the small ones, note that the average number of lines of a Smalltalk method is 7 [KLIM 96].

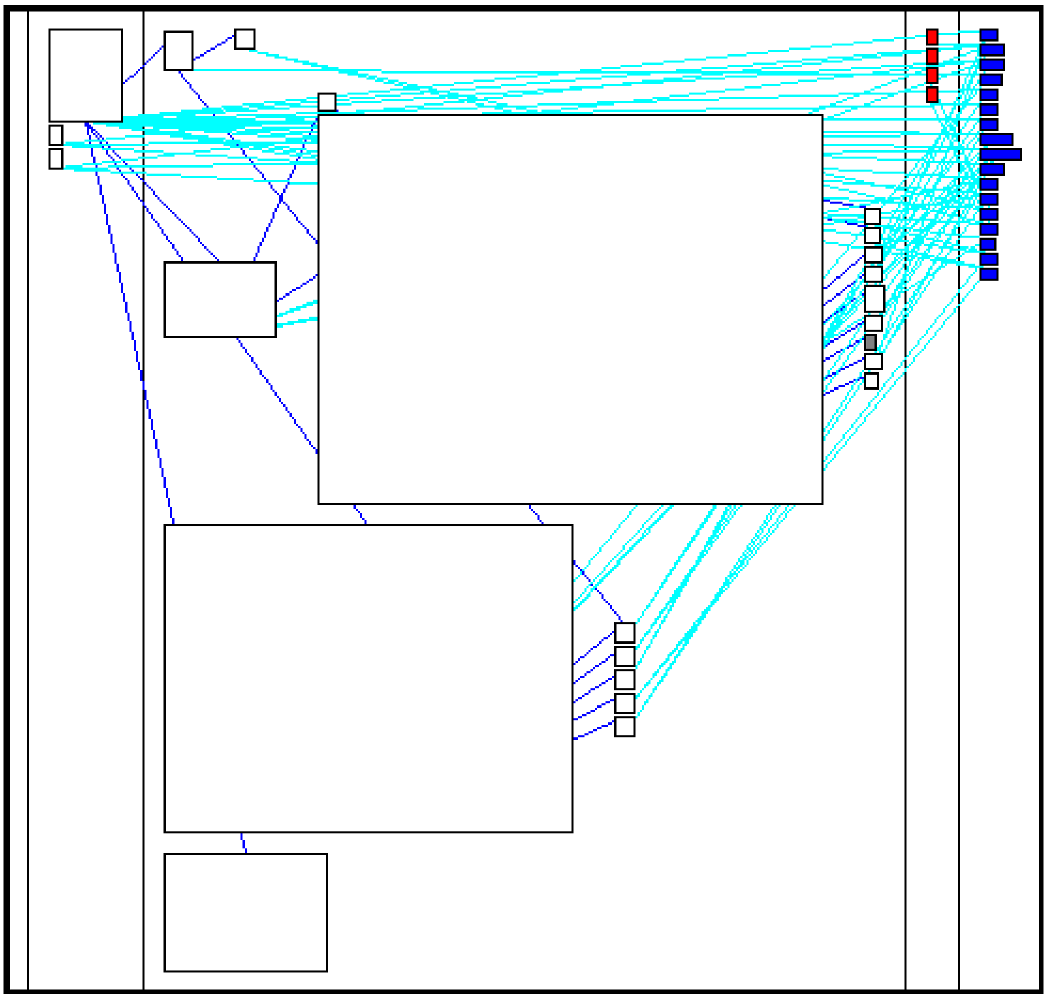


Figure 5.17: The *Method Clumps* pattern of the class *JunGNUDiff*.

Funnel

Graphically this pattern is composed of an inverse (right-to-left) tree of nodes whose root is on the right, forming a funnel. *Funnel* describes a group of methods that all converge towards a final functionality. It often occurs when a complex data structure is used that can be accessed by various interfaces. Identifying the final functionality is often the key to understand how data abstraction is used in the class. In addition, in Smalltalk metaclasses providing multiple examples or initialization possibilities exhibit this behavior. Figure 5.18 and the bottom blueprint in Figure 5.10 present *Funnel* patterns.

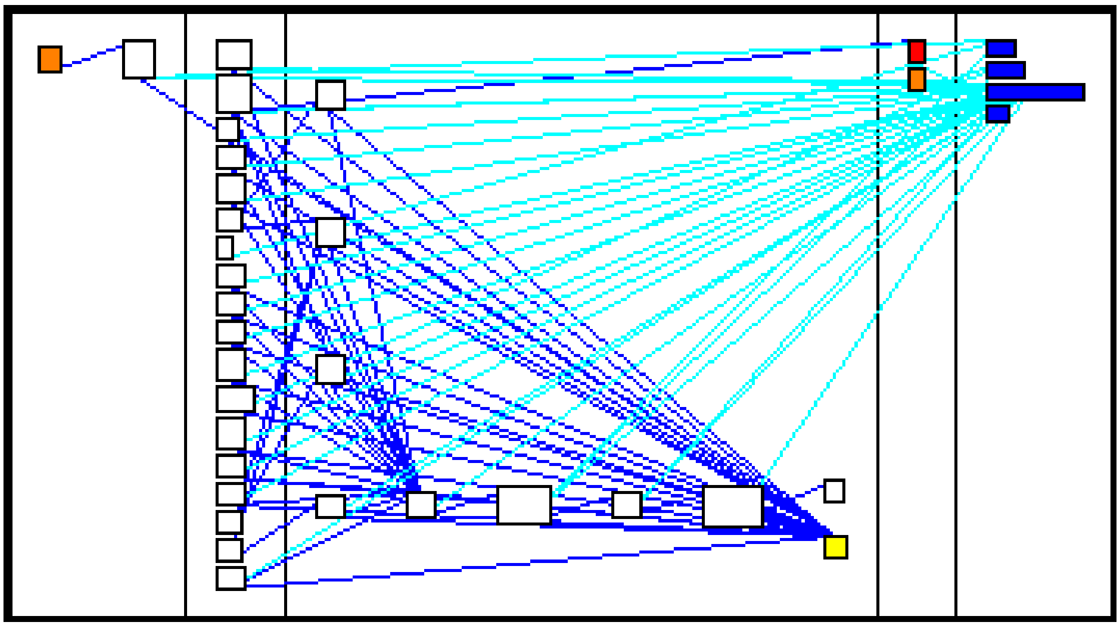


Figure 5.18: A *Funnel* pattern in the class blueprint of the class *JunMovieHandle*.

5.6.5 State Usage-based Blueprint Patterns

The way the attributes of a class are accessed by the methods creates patterns that provide important semantical information about the class. Three highly identifiable and recurrent blueprints occur: *Sharing Entries*, *Splittable State*, and *All State*.

Sharing Entries

Graphically the attribute nodes are accessed uniformly by groups of method nodes. This pattern represents the fact that multiple methods access the same state. Therefore it reveals a certain cohesion of the class regarding its state management. An example of such a pattern is emphasized in Figure 5.19 where nearly all methods access the third attribute from the top. Figure 5.15 presents *Sharing Entries* patterns as the two groups of method forming the *Single Entry* pattern access the same state.

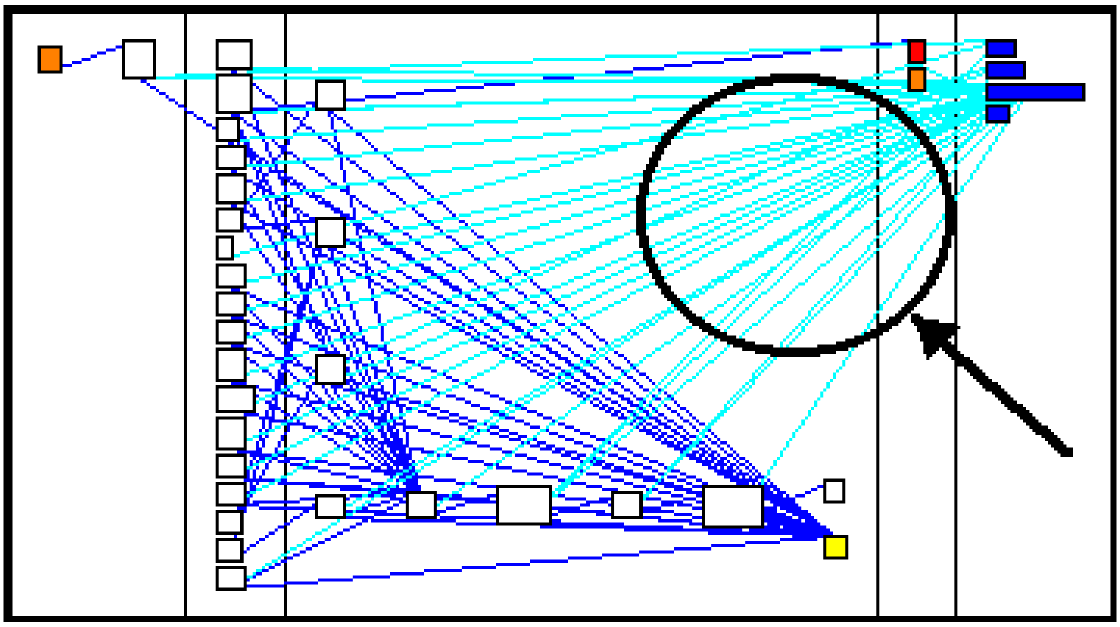


Figure 5.19: A *Sharing Entries* pattern in the class blueprint of the class *JunMovieHandle*.

Splittable State

Graphically this pattern presents two, rarely more, clearly separated groups of method nodes accessing two distinct set of attribute (blue) nodes. It occurs when a class is defined around several groups of methods each accessing only a subset of the class state. Classes presenting this pattern are showing a low cohesion and may be split if necessary. This pattern occurs with classes such as user interface classes, whose main purpose is to group together independent classes. *Splittable State* is rare, we included it in this section (and not in the section on rare blueprints) because it complements the other two blueprints presented here. We could not find one in the Jun case study, therefore omit a figure.

All State

Graphically this pattern presents groups of method nodes that have edges arriving to *all* the blue attribute nodes. It is semantically orthogonal to the other two and describes the fact that a group of methods accesses *all* the attributes of a class. When the class presents a *Single Entry* it often presents also the *All State* blueprint. The inverse is not true. Figure 5.20 shows an example where we see that all the attributes in the class are accessed by the two methods annotated as *A*. This remarkable behavior is also exhibited by the Figure 5.6 and the bottom blueprints in Figure 5.24.

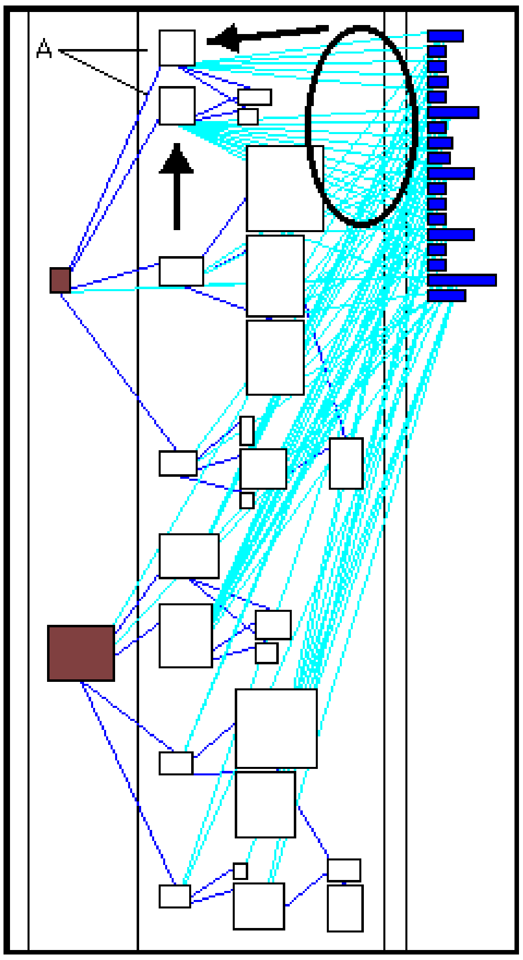


Figure 5.20: The class blueprint of the class *JunBmpImageStream* with an *All State* pattern.

5.7 Blueprint Patterns in the Context of Inheritance

The blueprints support class understanding within the context of their inheritance hierarchy. Within hierarchies some specific and recurrent patterns occur as well.

Micro Specializer

Graphically this pattern shows a small class blueprint composed of a couple of short methods, *i.e.*, mostly small brown or orange nodes. It denotes a small class which defines overriding and/or extending methods. Such classes are mainly used to specialize well identified behavior and they benefit from the structure and behavior of their superclasses. In Figure 5.8 we see some examples of the *Micro Specializer* blueprint.

Siamese Twin

This pattern is special because it is based on the similarity between two or more blueprints of sibling classes, in terms of methods, attributes, method invocations, and attribute accesses. This happens when the programmer forgot to refactor the common functionality into the superclass of the siamese twins or when the superclass implements complex logic that should be extended in a similar way in the subclasses. The three subclasses in Figure 5.21 are siamese twins, especially the one on the left and the one on the right override exactly the same methods. The bottom blueprints in Figure 5.24 present two large *Siamese Twin* patterns, which is rare in this size.

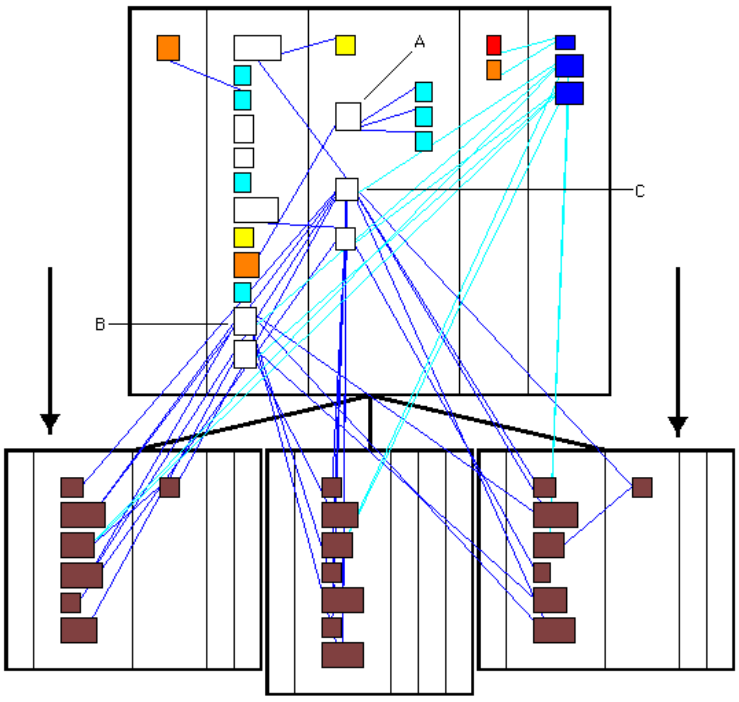


Figure 5.21: A class blueprint visualization of an inheritance hierarchy with the class *JunColorChoice* as root class. Each subclass contains a pure *Siamese Twin* pattern.

Island

Graphically this pattern presents a class blueprint without any edge going out or coming from other class blueprints. *Island* reveals classes that do not communicate with their superclasses, sibling classes, or subclasses. The communication between the class and its superclass is only performed via the template methods of the superclass. Note that such a class can also define new methods and new attributes. In Figure 5.22 we see that the subclass does neither invoke methods nor access attributes of its superclass. Furthermore we see that the subclass does neither override nor extend any methods of the superclass, since this would be visible as brown or orange method nodes: the subclass does not communicate with its superclass.

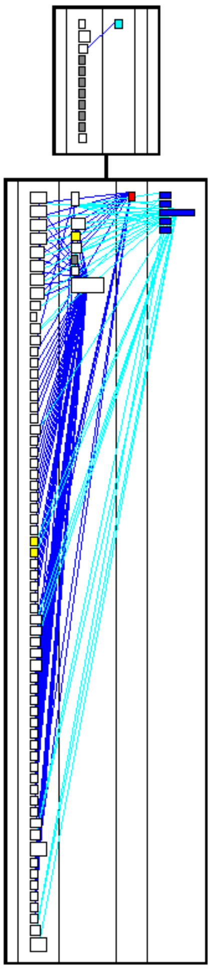


Figure 5.22: The subclass in this case shows an *Island* pattern, as it does not communicate at all with its superclass.

Adder / Extender / Overrider

Graphically this pattern presents class blueprints that are mainly white (adding), orange (extending), or brown (overriding). These patterns present the way classes add, extend, or override inherited behavior. The weight of these patterns, *i.e.*, the number of methods in one of these three colors compared with the total number of methods is an indication on the way the class fits within its inheritance hierarchy. The rightmost subclass in Figure 5.8 is a pure adder as it is completely white, while all the other subclasses denote heavy *Overrider* patterns, *e.g.*, have many overriding methods. None of these classes is extending superclass behavior. We also see that in Figure 5.23 the two subclasses are combinations of *Overrider* and *Adder* blueprints, denoted by the presence of several brown and white method nodes.

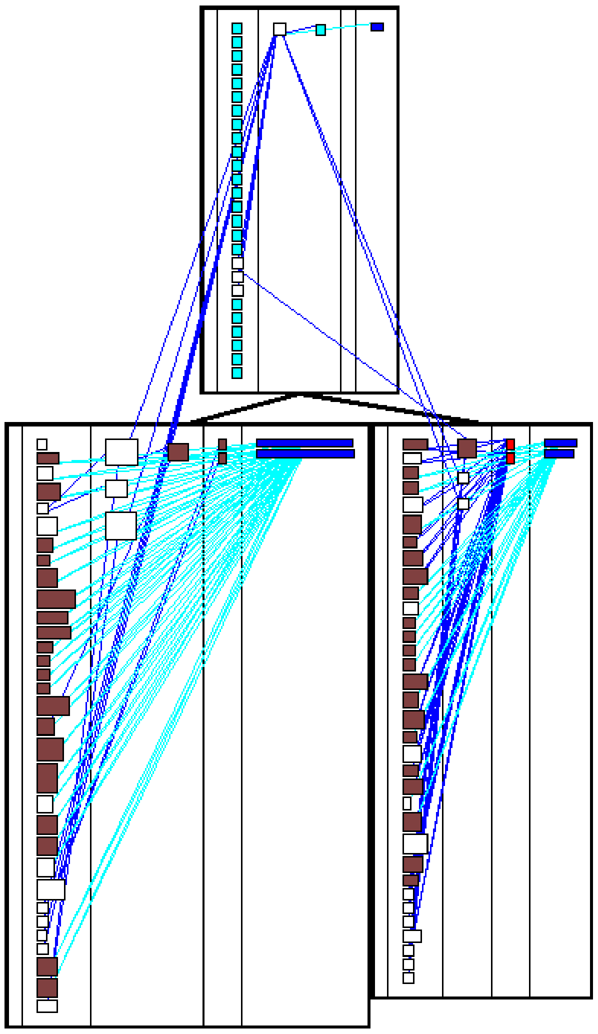


Figure 5.23: A nearly pure *Template* class blueprint of the root class *JunParametricSection*. The two subclasses are both heavy *Overrider* blueprints.

Template

Graphically this pattern shows a blueprint with a possibly small implementation layer and several cyan nodes, *i.e.*, abstract methods. It reveals that a class is not limited to an interface and that it defines some abstract methods. These classes are often mature classes. The class at the top of the hierarchy in Figure 5.7 is a good example of mature design: the class defines some template methods [GAMM 95] and abstract hook methods specializing the behavior inherited from its superclass. Figure 5.23 and Figure 5.5 both show a *Template* pattern.

5.8 Analysis of Blueprint Patterns

As we saw in the preceding sections, a pattern reveals a specific aspect of a class. Such an aspect can be then *qualified* depending on its presence in the class and/or *combined* with other patterns. Based on our experience we learned that the two pattern qualifications that convey the most semantics are when a blueprint is *pure*, *i.e.*, the class blueprint contains only one single pattern and when the blueprint is *heavy*, *i.e.*, a certain pattern has a strong presence in the class.

The fact that a class exhibits a single or several patterns acts as a reinforcing action for the understanding of the class, its role and also often its quality in terms of coding conventions or design. Pattern combinations are often the logical result of good design practices and presenting them here does not represent something new *per se*. However, in the context of class understanding they are an important validation of our approach showing that the patterns reveal class design and support a quick understanding of classes. During our experiences identifying one of the following combinations always accelerated the generation and validation of the hypothesis relative to the role or design of the subject class.

5.8.1 Frequent Blueprint Pattern Combinations

Single Constant Definer

Graphically this combination is a blueprint with one grey node. This combination (*Single* and *Constant Definer*) occurs often in the context of rich and important hierarchies. It represents a class consisting of only one method which returns a constant value. The location of the class within the hierarchy gives complementary information. This pattern often appears in two contexts:

- When the class is in the middle or top of the hierarchy, it reveals a default value shared by a large number of classes. This default value is often used as a way to discriminate over the type of an object in the form of methods named *isOfTypeXY()*, (*e.g.*, is2DExtrapolator, is3DExtrapolator, isVRML97..., etc.). Looking for the senders of such testing methods may reveal hidden explicit type-checks and therefore a lack of polymorphism and late-binding use in the system [DEME 02].
- When occurring on leaf classes, a *Single Constant Definer* often represents the definition of constants that specify hook methods of more complex schema as for example the various tags of the HTML language in the context of HTML generation.

This combination is efficient because on one hand it allows us to identify quickly the methods that could be misused and thus compromise the design of an application and on the other hand it shows a good use of object-oriented programming in the sense of reuse and customization.

Funnel, Wide Interface,* and *Accessor User

A *Funnel* is often a sign of good practice such as code functionality reuse and decomposition. Combined with *Wide Interface* and *Accessor User* it is clearly the sign of good decomposition and encapsulation as all the accesses to the underlying representation go via disciplined access to instance state. Figure 5.18 presents a *Wide Interface* combined with a *Funnel*.

Large Implementation* and *Wide Interface

Unfortunately not all the classes present aesthetic blueprints. This is especially true in the case of legacy systems where the system went through several years of maintenance and *ad-hoc* evolution. One of the most popular pattern combination is *Large Implementation* and *Wide Interface*, which has some commonalities with Riel's definition of a *god class* [RIEL 96].

Delegate* and *Wide Interface

Obviously when a class delegates methods to another class its interface includes more methods thus stressing the interface layer of the blueprint. Figure 5.5 and the class *B* in Figure 5.8 show this pattern combination.

Single Entry and *Large Implementation*

Frequently classes with a large implementation layer structure also their functionalities as *Single Entry*, *i.e.*, having a simple interface but a deep and narrow invocation tree. This happens often for scanners and parsers as their internal behavior is logically decomposed. In Figure 5.6 we see that the second node in the external interface layer is the entry point to a cluster of methods that represents this combination.

Siamese Twin and *Micro Specializer*

This combination is really frequent in rich hierarchies where the leaf classes take advantage of the abstraction and templates specified in the superclasses. The superclasses structure well-defined templates for functionalities that their subclasses then have just to specify. The subclasses in Figure 5.7 are such combinations.

5.8.2 Rare Blueprint Pattern Combinations

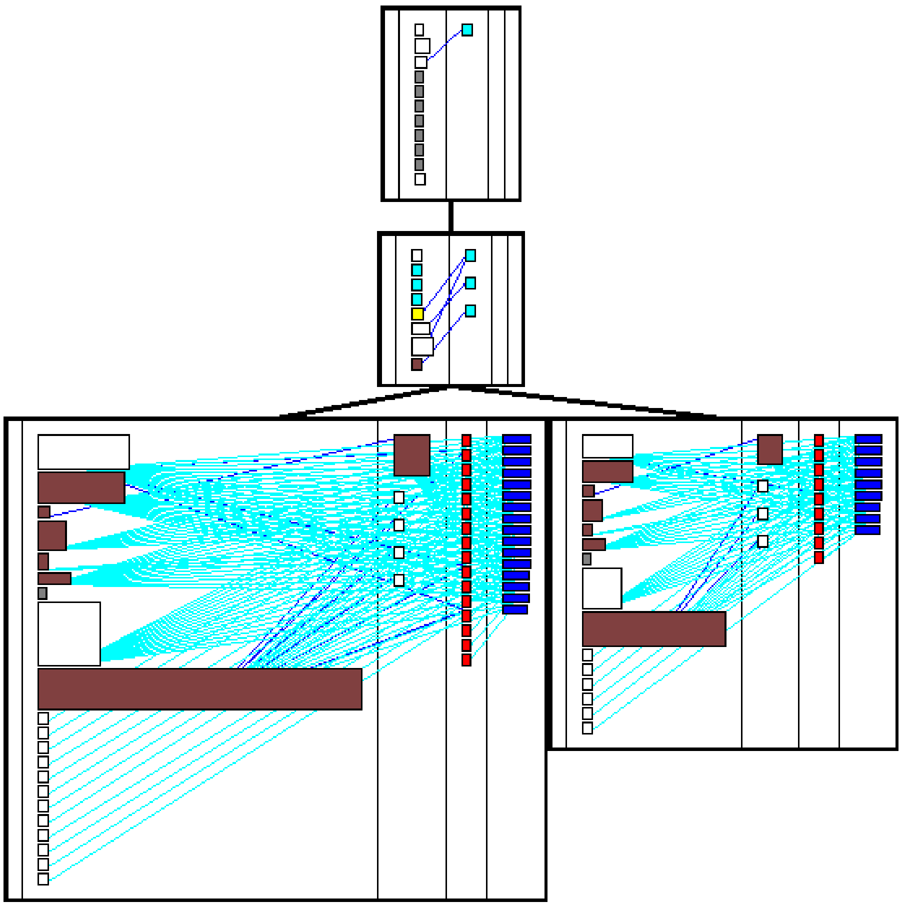


Figure 5.24: A nearly pure *Interface* class blueprint of the root class *JunGeometry*. The first subclass is a nearly pure *Template* blueprint, while the two leaf classes are combinations of the *Overrider*, and *All State* blueprint. Moreover they are *Siamese Twin* blueprints compared with each other.

Pure *Interface*

Graphically this pattern presents a blueprint having only methods in the interface layer. The root class in Figure 5.24 is a nearly pure *Interface* blueprint (with only one method in the implementation layer). This pattern may occur in the following cases:

- As a leaf class with overriding methods a pure *Interface* can be an incarnation of the NullObject design pattern which only defines default behavior that is most of the time absorbing messages or returning default values [WOOL 98].
- As a class near the top of an inheritance hierarchy, a pure *Interface* reveals a class only implementing method signatures. In such a situation the pure *Interface* normally defines abstract methods. A pure *Interface* without abstract methods is definitively a place to look further to see why none of the methods is abstract.

Pure *Interface* and *Constant Definer*

This combination represents a class whose methods are only returning constants. When a pure *Interface* is also a *Constant Definer*, it reveals that the class mainly represents a common set of values. In Smalltalk metaclasses exhibit this combination as there is no explicit constant definition via CONST-like constructs. The root class in Figure 5.24 is a good example of this combination.

Pure *Overrider*

As we mentioned earlier, an *Overrider* is often combined with a *Micro* or a *Siamese Twin* pattern. Pure overriders are rare because this implies that the superclass achieves a good level of design and that the domain allows one to structure subclasses by only defining a given set of methods. They reveal that the abstraction defined in the superclasses has been designed cautiously and can be reused by just redefining a precise set of methods. They are a good sign regarding the quality of the hierarchy. By looking manually at the overridden methods and checking whether all the sibling classes override consistently these methods allows one to focus on the class variation points and understand the key behavior of the classes. Each of the three subclasses shown in Figure 5.7 is a pure *Overrider*.

Extender and *Overrider*

From our experience, patterns present less signs related to the extension of a functionality than its redefinition except for instance initialization phases. Our hypothesis is that extension is a more complex act than a local redefinition. The combination of these two is then rare and often points to a mature design.

5.9 Tool Support: CodeCrawler and Moose

To obtain the class blueprint visualizations we use CodeCrawler as visualization tool and Moose as meta-model and provider of the metrics and semantic information (see Appendix A). Our tool CodeCrawler supports the synergy between opportunist reading of the code and the visualization of classes in the following ways:

- **Interactivity.** The blueprint visualizations do not merely represent source code, as in the case of static visualizations (*e.g.*, static pictures which cannot be manipulated), but they support direction manipulation. When the proposed layout does not suit the reengineer wishes, he can select, move, or delete connected, recursively connected, or not connected nodes.
- **Code Proximity.** At any moment the reengineer can access the code by clicking on any node and see the corresponding definition at the level of a method, at the level of the class, and using code browsers presenting superclasses and subclasses. Moreover he has the possibility to see permanently a floating window showing the code of the node over which the mouse pointer is passing.

5.10 Related Work

Among the various approaches to support reverse engineering that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids [PRIC 93] [Sta 98].

Many tools make use of static information to visualize software, like Rigi [TILL 94], Hy+ [CONS 93] [MEND 95], SeeSoft [EICK 92], Dali [KAZM 99], ShrimpViews [STOR 95], TANGO [STAS 90], as well as commercial tools like Imagix [3] to name but a few of the more prominent examples. However, most publications and tools that address the problem of large-scale static software visualization treat classes as the smallest unit in their visualizations. There are some tools, for instance the FIELD programming environment [REIS 90] or Hy+ [CONS 93] [MEND 95] which have visualized the internals of classes, but usually they limited themselves to showing method names, attributes, etc. and use simple graphs without added semantical information.

Substantial research has also been conducted on runtime information visualization. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [LANG 95], Jinsight and its ancestors [PAUW 93] [PAUW 99], Graphtrace [KLEY 88] or [RICH 99]. Various approaches have been discussed like in [JERD 97] where interactions in program executions are being visualized, to name but a few.

Nassi and Shneiderman proposed flowcharts to represent in a more dense manner the code of procedures [NASS 73]. Warnier/Orr-diagrams allow us to describe the organization of data and procedures [HIGG 87]. Both approaches only deal with procedural code and control-flow. Cross *et al.* defined and validated the effectiveness of Control Structure Diagrams (CSD) [CROS 98] [HEND 02], which is a graphical representation that depicts the control-structure and module-level organization of a program. Even if CSD has been adapted from Ada to Java, it still does not take into account the fact that a class exists within an hierarchy and in presence of late-binding.

We provide a visualization of the internal structure of classes in terms of their implementations and in the context of their inheritance relationships with other classes. In this sense our approach proposes a new dimension in the understanding of object-oriented systems.

5.11 Conclusion

As in object-oriented programming, classes are the primary abstractions based on which applications are built, we focus on supporting the reengineer to understand the internal structure of classes and how class behavior is developed in the context of the inheritance hierarchy in which it is defined. Our approach is based on the synergy between the class blueprint visualization and opportunist code reading [LITT 96] as the visualization helps building hypothesis, raising questions that a selective code reading verifies. As such it supports an understanding at multiple levels of abstractions [VON 96].

5.11.1 Summary

In this chapter we have presented the *class blueprint*, a polymetric view targeted at the understanding of classes and class hierarchies. The class blueprint visualizes the internal structure of classes, *e.g.*, an augmented call-graph enriched with metrics and semantic information. The class blueprint permits to identify patterns that help to understand the structure of classes. We have identified and described several of these patterns. Furthermore we have used the class blueprints on a case study and have discussed and verified our findings.

5.11.2 Benefits

The main benefits of our approach are the following:

[3]see http://www.imagix.com

- *Reduction of complexity.* Using a visualization named *class blueprint* we can make assumptions about a class without having to read the whole source code. This "taste" of the class, which conveys the purpose of a class, appears in two contexts: the class in isolation and the class within its inheritance hierarchy.

- *Identification of key methods.* The class blueprint by condensing the class stresses, some of its aspects. Based on the resulting signs shown by the blueprint, the reengineer builds hypotheses and gains insights on the structure and internal implementation of a class. The blueprint helps to *select the relevant* methods whose reading validates or invalidates the hypotheses of the reengineer.

- *A common vocabulary.* The recurrent patterns created by the blueprints define a common vocabulary for the class. This vocabulary supports the communication between reengineers during a reverse engineering process, in a similar manner to design patterns that constitute a vocabulary for design solutions.

- *Programming style detection.* After the display of several blueprints, the observer starts to identify common patterns in different blueprints. These patterns reflect the programming style of the developer, *i.e.*, in some case studies we are able to recognize which developer wrote the blueprinted classes.

5.11.3 Limits

Our approach is limited in the following ways:

- *Layout Algorithm.* The approach presented here relies heavily on an efficient layout algorithm in terms of space and readability. Especially in the case of very large classes, *i.e.*, having hundreds of methods, it may happen that the only real statement we can make is that the class is large (the *Giant* blueprint). However, patterns often occur in such classes providing important pieces of information.

- *Functionality.* The blueprint of a class can give the viewer a "taste" of the class at one glance. However, it does not show the actual functionality the class provides. The approach proposed here is thus complementary to other approaches used to understand classes.

- *Collaboration.* We do not address collaboration aspects between classes for the time being.

- *Static Analysis.* The approach presented here does not make use of dynamic information. This means we are ignoring runtime information about which methods get actually invoked in a class. This is especially relevant in the context of polymorphism and switches within the code. In this sense the class blueprint can be seen as a visualization of every possible combination of method invocations and attribute accesses.

5.11.4 Future work

In the future we plan to investigate the following ideas:

- *Collaboration.* Apply the class blueprint view on classes which are not within the same inheritance hierarchy, but which collaborate with each other. However this can be complex because of the presence of late-binding and the possible high number of method invocations between classes.

- *Cognitive Science.* The visualization algorithm presented here and the methodology coming with it are both *ad hoc*, and build empirically on several years of experimentation. It shows little connection with research from the field of cognitive science. We would like to understand more deeply how our approach fits into more general approaches such as the one proposed by Ware [WARE 00], Bertin [BERT 74], and Tufte [TUFT 90] [TUFT 01].

- *Qualitative Empirical Analysis.* We would like to have an empirical usability analysis and qualitative validation of our approach by letting reverse engineers use our system and to collect their experiences. A second possibility we want to explore is to do a controlled experiment on the reverse engineering "efficiency" of two groups of users.

- *Quantitative Empirical Analysis.* We have already made a first quantitative validation of the class blueprints [LANZ 01a], where for two different case studies we have listed the number of different blueprints we have found. Since in the meantime our vocabulary has changed, we want to make such an experiment again with our refined categorization of blueprints. What we have already seen in the mentioned case study is that the frequency of certain blueprints heavily depends on the case study and the coding conventions that have been used.

- *Languages.* The proposed approach has been developed to be applicable to any class-based object-oriented language. We already visualized C++ and Java classes as blueprints. A first observation was that the language or the mapping between the language to the blueprint layers still influences the blueprint. We plan to identify the variation points by applying the visualizations to a number of other object-oriented languages.

- *Pattern Recognition.* The use of techniques from the field of image processing and pattern recognition could eventually be used to automatically recognize the blueprint patterns. We are currently investigating the possibility of a cooperation with a research group working in artificial intelligence.

Chapter 6

Evolutionary Software Visualization: The Evolution Matrix

6.1 Introduction

Coping with huge amounts of data is one of the major problems of software evolution research, as several versions of the same software must be analyzed in parallel. A technique which can be used to reduce this complexity is *software visualization*, as a good visual display allows the human brain to study multiple aspects of complex problems. Another useful approach when dealing with large amounts of data are *software metrics*. Metrics can help to assess the complexity of software and to discover software artifacts with unusual measurements.

Summary. In this chapter we present a polymetric view called *evolution matrix* [LANZ 01b] [1] that allows for an understanding of the evolution of classes within object-oriented software systems and the evolution of the systems themselves. Moreover the evolution matrix acts as a revealer of certain specific situations that occur during system evolution such as pulsating classes that grow and shrink during the lifetime of the system. We define a simple vocabulary to describe such specific behaviors. The intention is to build a vocabulary for software evolution. Note that even if the results we present are obtained on software systems written in Smalltalk, the approach presented here does not depend on a particular programming language, as our underlying metamodel is language-independent [DUCA 00, DEME 01]. With the evolution matrix view we want to achieve the following evolutionary reverse engineering goals:

1. Understand the evolution of object-oriented software systems in terms of size and growth rate.
2. Understand at which point in time classes have been introduced into a system and at which moment they have been removed.
3. See if there are patterns in the evolution of classes. Such patterns help to understand the condition of a class in a time perspective, *e.g.*, how resistant to software evolution processes is a class, is it changed with every release of a system, or are there classes which are virtually immune to software evolution?

Contributions. The contributions of this chapter are the following:

- The definition of the *evolution matrix*, a polymetric view which visualizes the evolution of the classes of a software system, and provides an understanding of the evolution of classes and of a whole system. The evolution matrix lets the viewer identify *evolutionary patterns* which help to understand (1) the evolution of the whole system, and (2) the evolution of single classes,

[1] This chapter is an extended version of the article *Understanding Software Evolution using a Combination of Software Visualization and Software Metrics*, published in the LMO 2002 Proceedings (Languages et Modeles á Objets), pp. 135 - 149, Hermes Publications, 2002.

- The definition of a vocabulary based on these evolutionary patterns detected using the evolution matrix view.

Structure of the chapter. The chapter is structured as follows: in the next section (Section 6.2) we present the evolution matrix view, and, based on that, a categorization of classes (Section 6.3). Afterwards we apply and discuss our approach on some case studies (Section 6.4). We conclude the chapter by discussing the benefits and limits of our approach, as well as related work. Finally, we give an outlook on our future work in this area.

6.2 The Evolution Matrix

In this section we present the polymetric view *evolution matrix*, which supports the understanding of the evolution of the classes of a software system. We discuss the view and then show an example matrix. At the end of this section we introduce a categorization of classes based on their visualization within the evolution matrix.

6.2.1 The Layout Algorithm of the Evolution Matrix

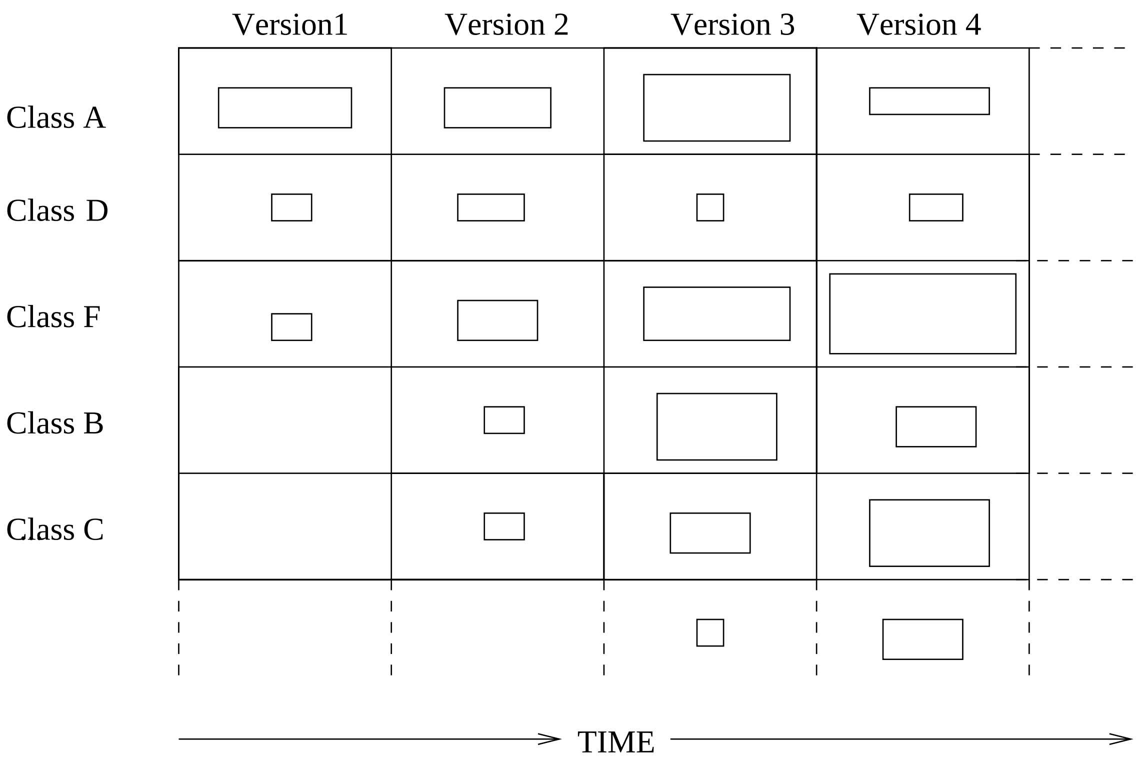


Figure 6.1: A schematic display of the Evolution Matrix. Classes A,D and F are alphabetically ordered and stay since version 1. Classes B and C appeared after version 2.

The evolution matrix displays the evolution of the classes of a software system. It has the following properties:

- Each column of the matrix represents a version of the software.
- Each row represents the different versions of the same class.

- Two classes in two different versions are considered the same if they have the same name.
- Within the columns the classes are sorted alphabetically in case they appear for the first time in the system. Otherwise they are placed at the same vertical position as their predecessors. This order is important because it allows one to represent the continuous flow of development of existing classes and stresses the development of new ones.

Figure 6.1 presents a schematic evolution matrix where the rows 5 and 6 represents new classes added in the system after the second release.

The evolution matrix allows us to make statements on the evolution of an object-oriented system at the system level. However, as the granularity at system level is too coarse, the evolution matrix is enhanced with additional information using metrics.

6.2.2 Characteristics at System Level

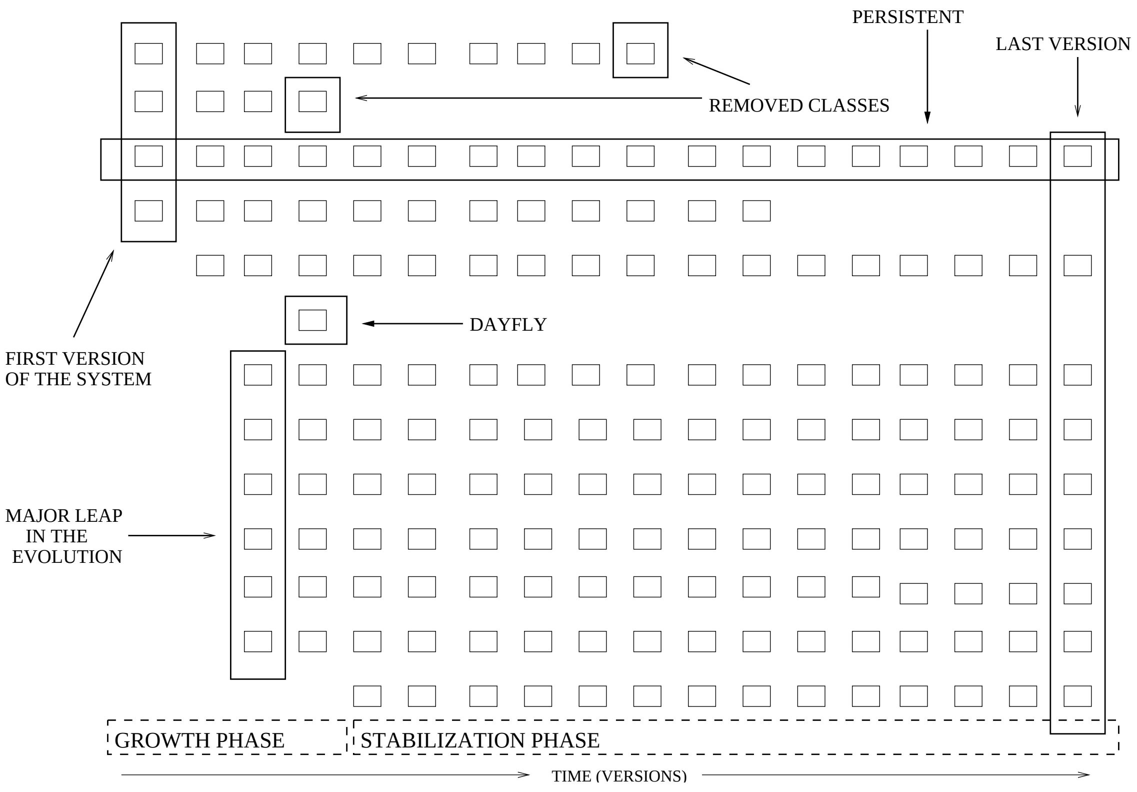


Figure 6.2: System level evolution aspects using the Evolution Matrix.

As we see schematically in Figure 6.2 at system level we are able to recover the following information regarding the evolution of a system:

- **Size of the system.** The number of present classes within one column is the number of classes of that particular version of the software. Thus the height of the column is an indicator of the system's size in terms of classes. The leftmost column is an indicator for the *initial* size of the system, while the rightmost column is an indicator for the *final* size of the system.
- **Addition and removal of classes.** The classes which have been added to the system at a certain point in time can easily be detected, as they are they are added at the bottom of the column of that version. Removed classes can easily be detected as well, as their absence will leave empty spaces on the matrix from that version on.

- **Growth and stabilization phases in the evolution.** The overall shape of the evolution matrix is an indicator for the evolution of the whole system. A growth phase is indicated by an increase in the height of the matrix, while during a stabilization phase (no classes are being added) the height of the matrix will stay the same. When a certain number of new classes are added they create a *leap phase.*

6.2.3 The Difference Evolution Matrix

The major drawback of the evolution matrix is that is renders absolute metric values, *e.g.*, the number of methods of a class, and since once is more interested in the changes between different versions of a class, we also visualize the relative values between different versions, *e.g.*, the number of added methods, in a slightly changed evolution matrix that we call *difference evolution matrix*. In Figure 6.10 we see such a difference evolution matrix which only renders as metric measurements the differences between subsequent versions of the classes. This has the advantage that growth phases are emphasized visually.

6.3 A Categorization of Classes based on the Evolution Matrix

We present here a categorization of classes based on the recurrent *patterns* we detect in the evolution matrix view. The categorization stems from the experiences we obtained while applying our approach on several case studies. A large part, but not all, of the vocabulary used here is taken out of the domain of astronomy. We do so because we have found that some of the names from this domain convey well the described types of evolution. This vocabulary is of utmost importance because a complex context and situation, like the evolution of software, can be communicated to another person in an efficient way. This idea comes from the domain of design patterns [GAMM 95].

During our case studies we have encountered several ways in which a class can evolve over its lifetime. We list here the most prominent types. Note that the categories introduced here are not mutually exclusive, *i.e.*, a class can behave like a *Pulsar* for a certain part of its life and then become a *White Dwarf* for the rest of its life. We first present the category which stems from the *presence* of a class within the matrix, and then the one which comes from the shape and the changes in shape of the classes.

6.3.1 Presence-based Patterns

Besides characterizing the evolution at system level, the evolution matrix provides some information about the classes themselves.

Two specific situations are worth being mentioned:

1. ***Dayfly* classes.** A *Dayfly* class has a very short lifetime, *i.e.*, it often exists only during one or two versions of the system. Such classes may have been created to try out an idea which was then dropped.

2. ***Persistent* classes.** A *Persistent* class has the same lifespan as the whole system. It has been there from the beginning and is therefore part of the original design. *Persistent* classes should be examined, as they may represent cases of dead code that no developer dares to remove as there is no one being able to explain the purpose of that class.

In Figure 6.3 we see schematic displays of these two patterns. The black nodes denote *Dayfly* patterns, while the gray nodes denote *Persistent* patterns.

Note that the system level view provided by the evolution matrix is not precise enough. Hence, a stabilization phase only describes the fact that classes stayed over multiple versions of the system. Nothing is said about the quality of the changes if any occurred. Such information is crucial for understanding a system, that is why the evolution matrix is enhanced using software metrics.

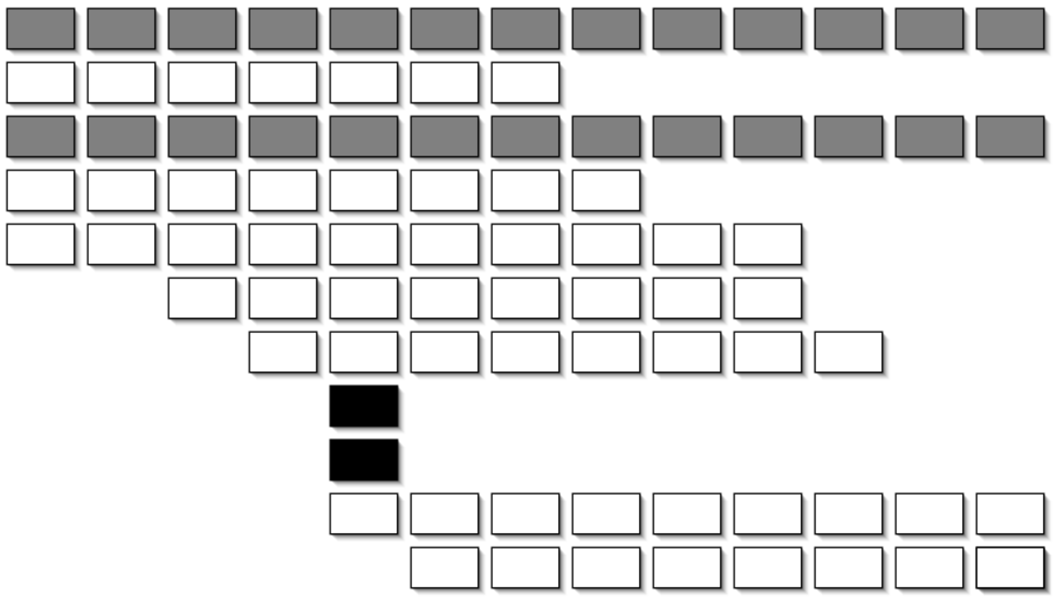


Figure 6.3: The visualization of *Dayfly* and *Persistent* classes.

6.3.2 Shape-based Patterns

Pulsar

A *Pulsar* class grows and shrinks repeatedly during its lifetime, as we see in Figure 6.4. The growth phases are due to additions of functionality, while the shrinking phases are most probably due to refactorings and restructurings of the class. Note that a refactoring may also make a class grow, for example when a long method is broken down into many shorter methods. *Pulsar* classes can be seen as hot places in the system: for every new version of the system changes on a *Pulsar* class must be performed.

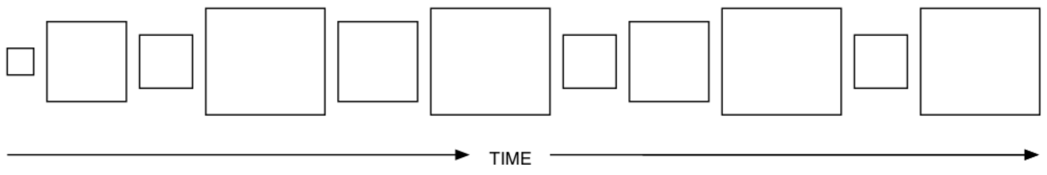


Figure 6.4: The Visualization of a *Pulsar* class.Note that the shape may change depending on the metrics associated with the representation.

Supernova

A *Supernova* is a class which suddenly explodes in size, and eventually becomes a *Red Giant* class. The reasons for such an explosive growth may vary, although we have already made out some common cases:

- Major refactorings of the system which have caused a massive shift of functionality towards a class.
- Data storage classes which mainly define attributes whose values can be accessed. Due to the simple structure of such classes it is easy to make such a class grow rapidly.

- So-called *sleeper* classes. A class which has been defined a long time ago but is waiting to be filled with functionality. Once the moment comes the developers may already be certain about the functionality to be introduced and do so in a short time.

Supernova classes should be examined closer as their accelerated growth rate may be a sign of unclean design or introduce new bugs into the system.

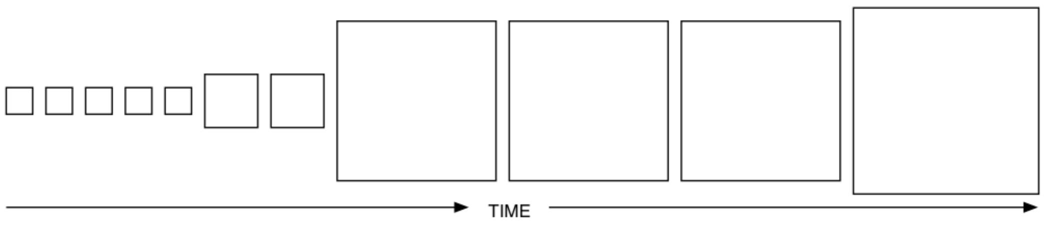


Figure 6.5: The visualization of a *Supernova* class.

White Dwarf

A *White Dwarf* is a class who used to be of a certain size, but due to varying reasons lost the functionality it defined to other classes, and slowly dwindles to become an *Idle* class. We can see a schematic display of a *White Dwarf* class in Figure 6.6. *White Dwarf* classes should be examined for signs of dead code, *i.e.*, they may be obsolete and therefore be removed.

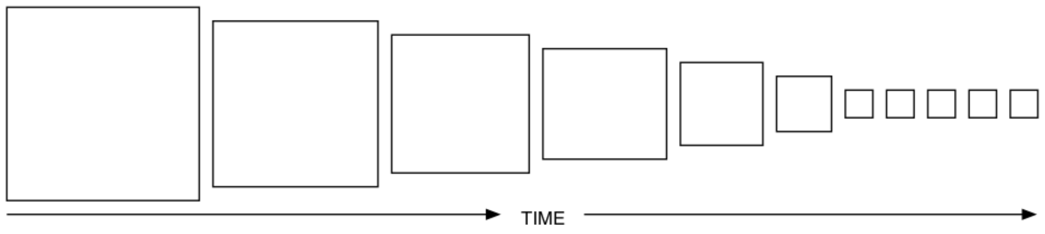


Figure 6.6: The visualization of a *White Dwarf* class.

Red Giant

A *Red Giant* class can be seen as a permanent god class [RIEL 96], which over several versions keeps on being very large. We can see a schematic display of a *Red Giant* class in Figure 6.7. God classes tend to implement too much functionality and are quite difficult to refactor, for example using a split class refactoring [FOWL 99].

Idle

An *Idle* class is one which does not change over several versions of the software system it belongs to. We can see a schematic display of an *Idle* class in Figure 6.8.

We list here a few reasons which may lead to an *Idle* class:

- Dead code. The class may have become obsolete at a certain point in time, but was not removed for varying reasons.

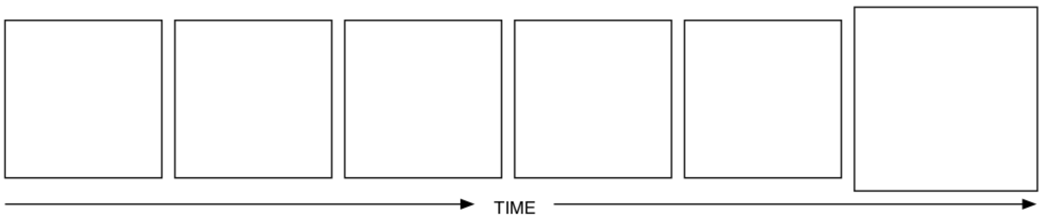


Figure 6.7: The visualization of a *Red Giant* class.

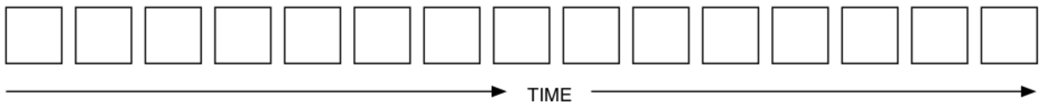


Figure 6.8: The visualization of an *Idle* class.

- Good design. *Idle* classes can have a good implementation or a simple structure which makes them resistant to changes affecting the system.
- The class belongs to a subsystem on which no work is being performed.

6.4 Illustration of the Approach

In this section we present two case studies whose evolution we have visualized using the approach described above. We shortly introduce each case study, and then show and discuss them.

6.4.1 MooseFinder

MooseFinder [STEI 01] is a small to medium sized application written in VisualWorks Smalltalk by one developer in little more than one year as part of a diploma thesis. We have taken 38 versions of the software as a case study.

Discussion. In Figure 6.9 we can see the evolution matrix of MooseFinder. We see that the first version on the left has a small number of classes and that of those only few survived until the last version, *i.e.*, are *Persistent* classes. We can also see there have been two major leaps and one long phase of stabilization. Note that the second leap is in fact a case of massive class renaming: many classes have been removed in the previous version and appear as added classes in the next version. There is also a version with a few *Dayfly* classes. The classes themselves rarely change in size except the class annotated as a renamed *Pulsar* class, which at first sight seems to be one of the central classes in the system.

The Difference Evolution Matrix. Figure 6.10 presents the same system where the difference metrics are represented. It reveals even more the sudden increases in size of certain classes. The interesting property of this view is that emphasizes changes. For example, having two flat boxes following each others shows that the class grows over the two versions. This view allows also to see where attributes have been added, *e.g.*, graphically this would mean to have nodes which are taller than the minimal node height.

6.4.2 Supremo

Supremo [KONI 01] is also written in VisualWorks Smalltalk. We have taken 21 versions of this application as a case study.

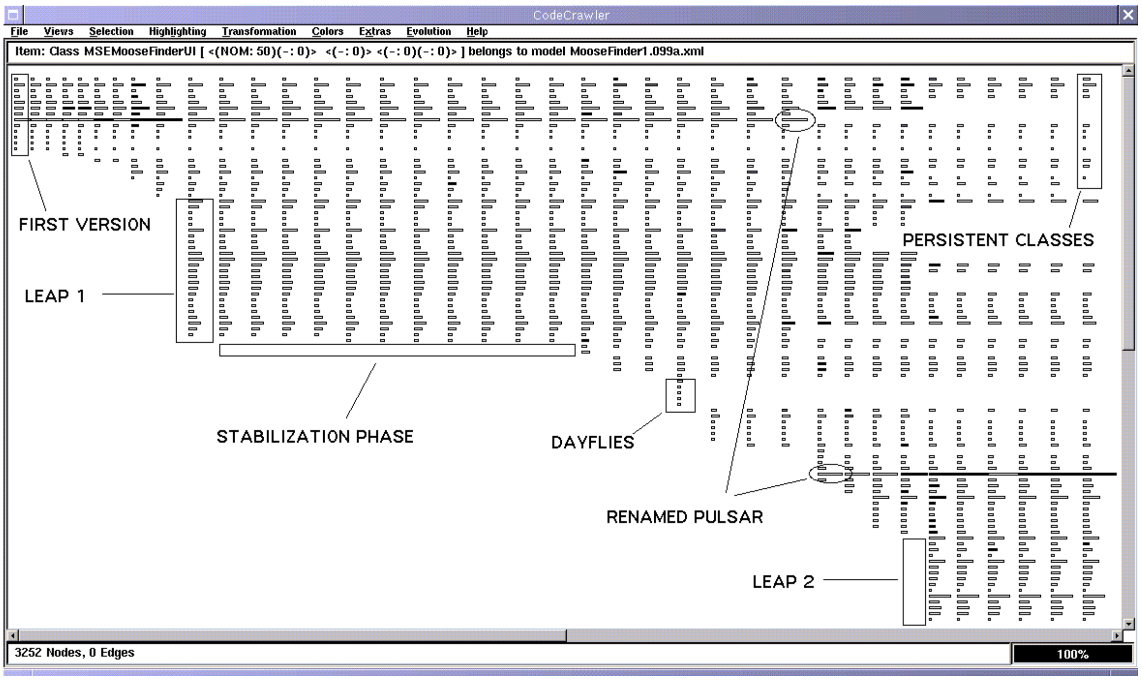

CodeCrawler
File Views Selection Highlighting Transformation Colors Extras Evolution Help
Item: Class MSEMooseFinderUI [<(NOM: 50)(-: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model MooseFinder1.099a.xml
FIRST VERSION
LEAP 1
STABILIZATION PHASE
DAYFLIES
RENAMED PULSAR
PERSISTENT CLASSES
LEAP 2
3252 Nodes, 0 Edges
100%

Figure 6.9: The Evolution Matrix of MooseFinder.

Discussion. In Figure 6.11 we see the evolution matrix of Supremo. We see that there is apart from a stabilization phase a constant growth of the system with three major growth phases. Note that the last growth phase is due to a massive renaming of classes. There are several *Pulsar* classes which strike the eye, some of which have considerable size. We can also see that from the original classes only two are *Persistent*, *i.e.*, the whole system renewed itself nearly completely. Figure 6.12 presents the same system using the differential view which emphasizes the changes made.

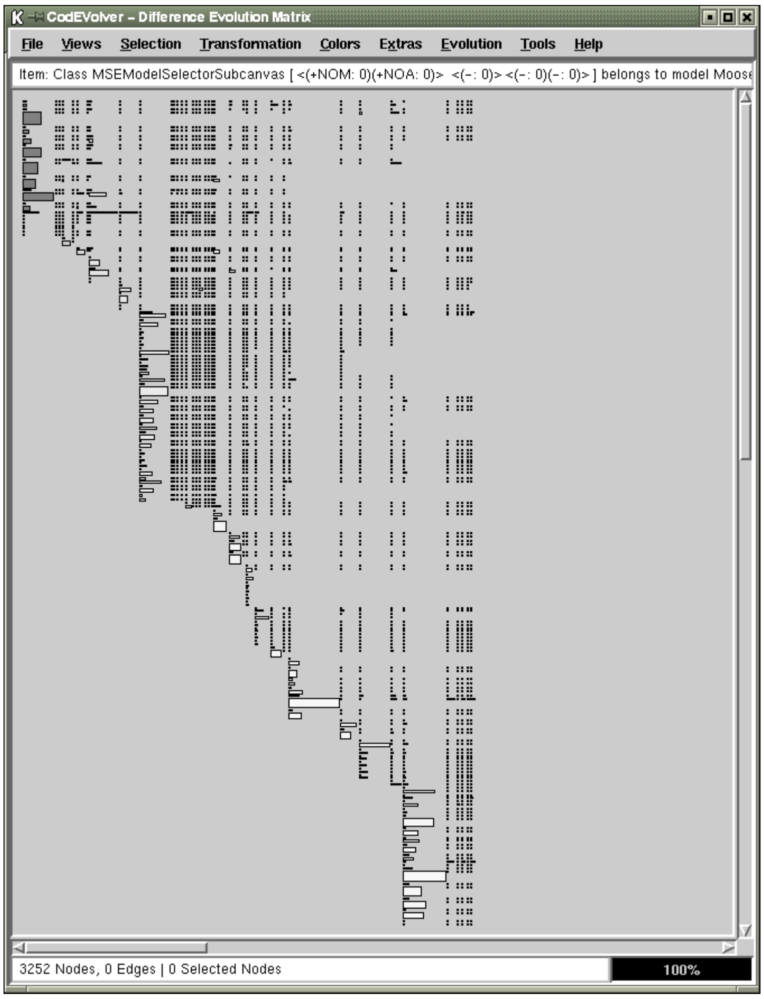
6.5 Tool Support: CodEVolver

In order to obtain the evolution matrix view we had to extend CodeCrawler, especially by providing the loading facilities, *i.e.*, we had to put in place a mechanism which allows us to load several versions of the same software in parallel, and then get the necessary information by querying the different models. This extension of CodeCrawler called CodEVolver (see screenshots in this chapter) integrates neatly with CodeCrawler. We describe and discuss the architecture of CodeCrawler, and the way it can be extended as in this case, in Appendix A.

6.6 Related Work

Among the various approaches to understand software evolution that have been proposed in the literature, graphical representations of software have long been accepted as comprehension aids. Holt and Pak [HOLT 96] present a visualization tool called GASE to elucidate the architectural changes between different versions of a system.

Rayside *et al.* [RAYS 98] have built a tool called JPort for exploring evolution between successive versions of the JDK. Their intent was to provide a tool for detecting possible problem areas when developers wish to port their Java tools across versions of the JDK. They provide evolution analysis at the level of Reuse Contracts [STEY 96].



Item: Class MSEModelSelectorSubcanvas [<(+NOM: 0)(+NOA: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Moose

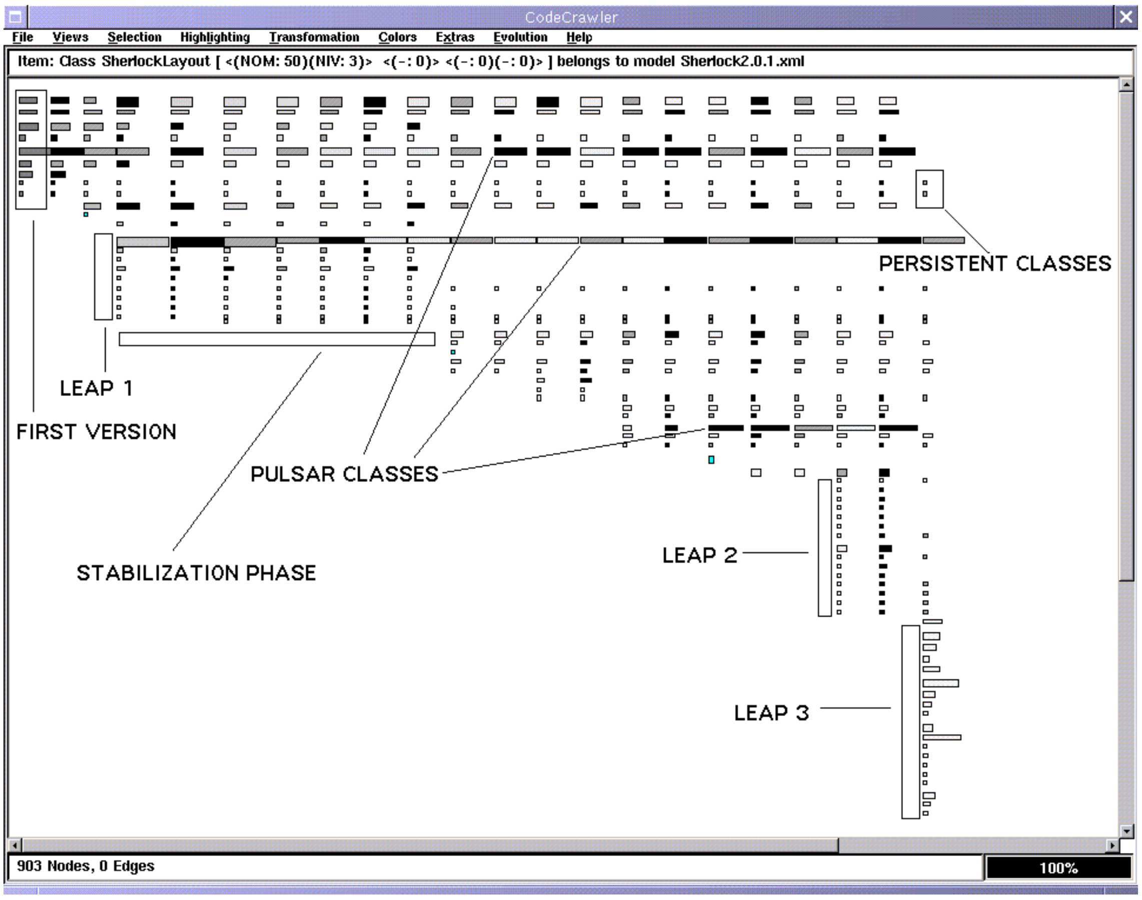
3252 Nodes, 0 Edges | 0 Selected Nodes

Figure 6.10: The Difference Evolution Matrix of MooseFinder.

In [JAZA 99, RIVA 98] Riva presents work which has similarities with ours, *i.e.*, they also visualize several versions of software (at subsystem level) using colors. Through the obtained colored displays they can make conclusions about the evolution of a system. Their approach differs as they do not have actual software artifacts but only information about software releases. This implies that they cannot verify the correctness of their informations. Our approach allows us to enrich the display using metrics information as well as being able to access every version of the software artifacts.

Burd and Munro have been analyzing the calling structure of source code [BURD 99]. They transformed calling structures into a graph using dominance relations to indicate call dependencies between functions. Dominance trees were derived from call-directed-acyclic-graphs [BURD 99]. The dominance trees show the complexity of the relationships between functions and potential ripple effects through change propagation.

Gall and Jazayeri examined the structure of a large telecommunication switching system with a size of about 10 MLOC over several releases [GALL 97]. The analysis was based on information stored in a database of product releases, the underlying code was neither available nor considered. They investigated first in measuring the size of components, their growth and change rates. The aim was to find conspicuous changes in the gathered size metrics and to identify candidate subsystems for restructuring and reengineering. A second effort on the same system focused on identifying logical coupling among subsystems in a way that potential structural shortcomings could be identified and examined [GALL 98].



Item: Class SherlockLayout [<(NOM: 50)(NIV: 3)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Sherlock2.0.1.xml

903 Nodes, 0 Edges

Figure 6.11: The Evolution Matrix of Supremo.

Sahraroui *et al.* [SAHR 00, LOUN 98] present another aspect of the research on software evolution which is the prediction of the evolution. Our current focus is to understand the evolution even if our long term goal is to gain a better prediction on which parts of the system will cause problems.

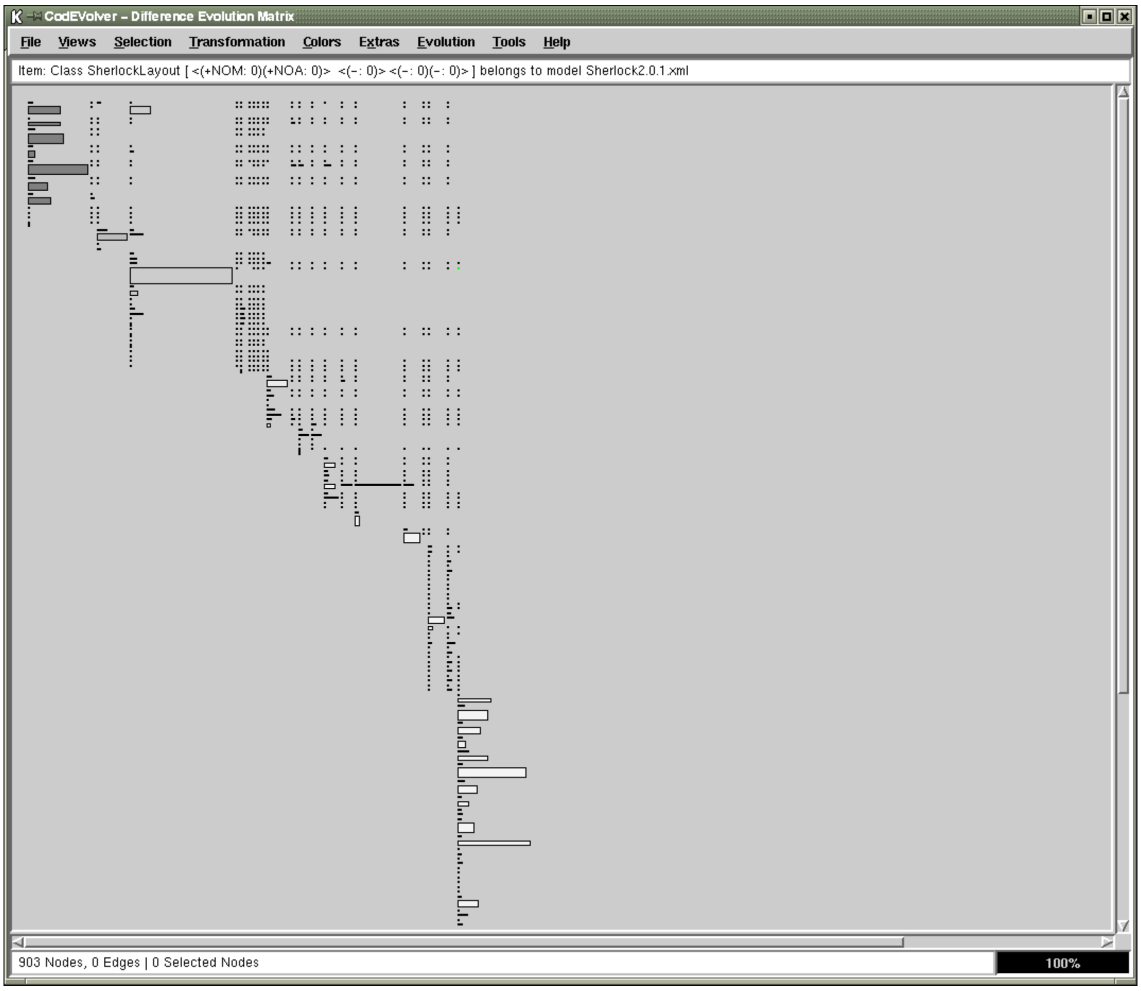
6.7 Conclusion

We presented an approach for helping the understanding of system evolution which is based on the *evolution matrix*, a polymetric view which displays classes on a two-dimensional matrix and enriches the visualization of the classes with metrics information. The evolution matrix can thus compress the vast amount of information contained in the evolution of a software system into a view which can be grasped in one glance. The matrix permits to observe the evolution of the whole system (by looking at the shape of the matrix) and the evolution of single classes (by looking at how the classes evolve from left to right).

6.7.1 Benefits

The evolution matrix helped us to provide answers to the evolutionary reverse engineering goals we have set:

- It helped us to understand the evolution of object-oriented software systems in terms of size and growth rate.


CodEVolver – Difference Evolution Matrix
File Views Selection Transformation Colors Extras Evolution Tools Help
100%

Item: Class SherlockLayout [<(+NOM: 0)(+NOA: 0)> <(-: 0)> <(-: 0)(-: 0)>] belongs to model Sherlock2.0.1.xml

903 Nodes, 0 Edges | 0 Selected Nodes

Figure 6.12: The Difference Evolution Matrix of Supremo.

- It helped to understand at which point in time classes have been introduced into a system and at which moment they have been removed.
- It let us detect patterns in the evolution of classes.

6.7.2 Limits

The presented approach has the following limitations:

- It is fragile regarding the renaming of the classes. Right now we consider a class similar to the subsequent versions if it has the same name. This assumption is too limitating and we plan to remove it by applying some simple heuristics to identify renamed classes such as a percentage of common methods and attributes.
- The view itself is not scalable in the sense that the evolution matrix of a very large system would fit on one screen only by zooming out, and this would decrease the interactivity, since we want to be able to interact with each version of each class. We think that the introduction of grouping techniques could increase the scalability of the evolution matrix.

6.7.3 Future Work

In the future we plan to investigate the following ideas:

- We would like to apply the evolution matrix at other levels of granularity. In particular, we want to be able to reason in terms of subsystems, packages or applications because these concepts represent conceptually linked classes in large applications. In such a context we would like to understand the evolution of subsystems, inside them and between them when for example a class is changing subsystem.
- Applying other metrics such the number of lines of codes in combination with the number of methods or statements in the class should be investigated to see if we can qualify the actual changes, *i.e.*, new methods can be added as the results of code refactoring while in the same time the number of lines can decrease.
- The choice of the case studies is also another factor that we would like to analyze. Indeed, the rates of changes may be quite different with longer periods between releases. In our experiences we have access to all the versions made by the developers and could not really assess major versions.
- we plan to apply the same approach to several versions of large systems like Squeak, Java Swing, VisualWorks Smalltalk and the Microsoft Foundation Classes (MFC) where the time spent between two versions can be months or years.

Chapter 7

Conclusions

In this chapter we summarize the contributions made in this dissertation, discuss the benefits of our approach, and point to directions for future work.

7.1 Contributions

In this dissertation we presented a new, lightweight approach to enrich software visualizations with metrics and other semantic information. We call these enriched visualizations *polymetric views*. The polymetric views are customizable and can be easily adapted to different contexts. We used the polymetric views in three different reverse engineering contexts, namely (1) coarse-grained software visualization, (2) fine-grained software visualization, and (3) evolutionary software visualization.

- *Coarse-grained software visualization.* We presented and discussed several polymetric views aimed at the first phase of a reverse engineering process. During that stage, the reverse engineer needs to build a mental image of the software system he is analyzing. We presented a reverse engineering approach based on clusters of polymetric views. Such an approach is needed, because the reverse engineer not only needs to know what he is currently looking at, but he also needs to know which are the next steps, *i.e.*, which part or aspect of the system he wants to understand comes next. Using the coarse-grained views we were able to reach the coarse-grained reverse engineering goals we had set up, namely:
  - Assess the overall quality of the system
  - Gain an overview of the system in terms of size, complexity, and structure.
  - Locate and understand the most important classes and inheritance hierarchies, *i.e.*, find the classes and hierarchies that represent a core part of the system's domain and understand their design, structure in terms of implementation, and purpose in terms of functionality.
  - Identify exceptional classes in terms of size and/or complexity compared to all classes in the subject system. These may be candidates for a further inspection or for the application of refactorings.
  - Identify exceptional methods in terms of size and/or complexity compared to the average of the methods in the subject system. These may be candidates for a further inspection regarding duplicated code or for the application of refactorings.
  - Locate unused, *e.g.*, dead code. This can be unused attributes, methods that are never invoked or that have commented method bodies, unreferenced classes, etc.
- *Fine-grained software visualization.* We presented and discussed the polymetric view *class blueprint*, a semantically augmented call- and access-graph of the methods and the attributes of classes. The class blueprint helps to understand and develop a mental image of the visualized classes. The class blueprint view provides the following benefits:

  - *Reduction of complexity.* It helps to make assumptions about a class without having to read the whole source code. This "taste" of the class, which conveys the purpose of a class, appears in two contexts: the class in isolation and the class within its inheritance hierarchy.
  - *Identification of key methods.* By condensing the informations contained in a class, it stresses some of its aspects. Based on the resulting signs shown by the blueprint, the reengineer builds hypotheses and gains insights on the structure and internal implementation of a class. The blueprint helps to *select the relevant* methods whose reading validates or invalidates the hypotheses of the reengineer.
  - *A common vocabulary.* The recurrent visual patterns created by the blueprints define a common vocabulary for the class. This vocabulary supports the communication between reengineers during a reverse engineering process, in a similar manner to design patterns that constitute a vocabulary for design solutions.
  - *Programming style detection.* After the display of several blueprints, the observer starts to identify common visual patterns in different blueprints. These patterns reflect the programming style of the developer, *i.e.*, in some case studies we are able to recognize which developer wrote the blueprinted classes.

- *Evolutionary software visualization.* We presented and discussed the polymetric view *evolution matrix*. It allows for an understanding of the evolution of classes within object-oriented systems and the evolution of the systems themselves. Moreover the evolution matrix acts as a revealer of certain specific situations that occur during system evolution such as pulsating classes that grow and shrink during the lifetime of the system. We defined a simple vocabulary to describe such specific behaviors. The evolution matrix view provides the following benefits:

  - It provides system wide views that help to understand essential changes during the evolution of an application.
  - It provides a finer understanding of the class evolution.
  - It builds a vocabulary to describe system and class evolution.
  - It scales well. However, we expect to have some screen limitation problems with huge systems. Working at another level of abstraction will be required.

We claimed that the polymetric views can help to greatly reduce the complexity of a reverse engineering process. Moreover, the added metrics and semantic information increase the amount of information that is visually transmitted to the viewer. The polymetric views support *opportunistic code reading, i.e.*, the goal of the polymetric views is not to replace code reading, but to point the viewer to locations of interest. To validate our claim we applied different polymetric views on several case studies, some of which are presented in this thesis, with the goal of understanding the subject systems.

7.2 Future Work

The future work pertinent to each one of the three different aspects we have covered in this thesis (coarse-grained, fine-grained, and evolutionary understanding) is discussed in the respective chapters.

In this section we present several ideas that could not be realized or implemented, largely due to a lack of time and a lack of human resources. The future work described here does not necessarily have a direct connection to what we discussed in this thesis. Nonetheless we would like to present it here:

- Software visualization, whose roots lie mainly in information visualization, is limited by this legacy. Most of the time ideas from information visualization are applied on static systems, for example websites, databases, and large documents. Software systems on the other hand are constantly being evolved by their developers and maintainers and can be regarded as living systems. This difference calls for a much higher degree of flexibility and above all interactivity which visualization tools must offer. The fact that most software developers still think in terms of *edit-compile-run* cycles may be one of the major reasons for the lack of success of software visualization tools which use mainly graphical approaches (as opposed to other visualizations, *e.g.*, textual enhancements like the use of color editing, different fonts, hypertext, etc.).

The future work would thus consist in developing highly interactive development environments which would make the distinction between forward and reverse engineering / reengineering superfluous.

- Visualization of software evolution is still a largely unexploited research field. This may be due to the fact that software evolution research is still focused on managing the huge sets of data and noise which denote this research topic. We think that an approach based on a *level of detail* (LOD), as used in 3D and Virtual Reality, *e.g.*, games like so-called first-person shooters, which allows one to increase the granularity of the observed information at will, may decrease the encountered problems and thus increase the quality of the information one wants to see.

  In this context we think that the use of animation, *i.e.*, moving pictures of evolving software systems, could delegate the task of noise reduction to the human eye and brain. On the other hand we are aware that animation on its own is not enough, because in software evolution the main interest does not lie in observation, but in comparing different versions. In the case of animations, the axis of (development) time would be mapped on the axis of (animation) time, which makes comparisons difficult, if not impossible. However, by adding the right information one could even profit from animation techniques, if not only to obtain a high-level view of the evolution of a complete software system. Here again the right information is the key to interesting results.

- As presented in Chapter 2 software visualization does not necessarily mean a visual display but can include other means as well. An interesting, yet under-exploited, research topic is software auralization, which visualizes software using sounds. It has mainly been used for rendering the dynamic behavior of software, *e.g.*, each time a program passes through a certain loop a sound is generated, or the pitch of the generated sound varies according to the state a program is in.

  Although it doesn't make much sense to use this technique for the auralization of the static structure of software, it could be used for rendering evolving software. For example a sound could be generated whose pitch changes according to the size of the system. Several other paths could be explored in this topic, but once again this research is heavily dependent on having as much information, *i.e.*, version information of the examined system, as possible.

  Examples of such pieces of information are the commit-time, the developer's name tag, and of course the version number. Other important pieces of information are the developer's comments, their size and the content. Knowing when, how, and what a developer says about the system he's being developing could be interesting.

- Software evolution research depends on having consistent sets of information about evolving systems, *i.e.*, the granularity level of the available information heavily influences the quality of the research. Although interesting experiments have been conducted even with as little information as the version numbers of subsystems [RIVA 98], high-quality research depends on having as much information as possible. This is where versioning systems like CVS or RCS can provide a great deal of information, although the granularity level is still only of released versions and not at the level of micro-changes, for example the addition and removal of methods. Versioning systems should not be used as external tools as they are right now, but should be neatly integrated into development environments, in order to provide as much information about the evolution of software as possible.

  An example of such the information is the tracing of browsers, which allows one to observe the *behavior* of programmers during the forward and reverse engineering phases of a software system's evolution, as can already be done in the StarBrowser [WUYT] by the SmallBrother plugin. Recovering such information would however involve a tight integration with the development environment at hand. This again would involve an invisibility of such a plugin or extension, *i.e.*, the plugin must not disturb or change the behavior of the developers during their programming.

7.3 Closing Words

All too often research conducted in the field of software visualization, a descendant of information visualization, is ignoring the great insights obtained by people like Bertin and more recently Tufte, Ware, and Stasko and others. This generated in the last few years dozens of software visualization tools that mainly offer the same functionalities and introduce little new insights. We believe that in order for software visualization to become a respected, serious, and established research field, software visualization researchers should settle on a common benchmark which would allow one to accept and/or reject new/old ideas and to discuss these ideas more critically.

The greatest problem that software visualization faces, is that it is a unique conjunction of various aspects and techniques and requires from researchers a wide range of talents among which

- a deep sense of esthetics,
- a thorough knowledge in software engineering, software reengineering and software reverse engineering,
- a vast experience in programming languages and integrated development environments (IDEs),
- a solid talent in developing software to efficiently implement new ideas,
- a good basis in human-computer interaction (HCI).

Sadly, no teaching or research institution currently offers such a broad range of disciplines or even tries to aggregate them into an interdisciplinary course.

Chapter 8

Epilogue

Even though we believe that the contributions made in this thesis may be helpful, as long as there is no tight integration of software visualization tools with current development environments, only a little part of their benefits will flow into mainstream software development and its industry.

However, it will stay a promising and interesting research field because of its complexity, reality, and beauty.

In that sense: we are going to see...

Michele Lanza

Aprile 27, 2003

Appendix A

CodeCrawler - Implementation

A.1 Introduction

In this chapter we discuss the implementation of CodeCrawler [1]. CodeCrawler is the software visualization tool we have implemented during the past years and which made possible all the visualizations presented here. In this chapter we discuss the implementation of CodeCrawler according to five aspects and then generalize the lessons learned into design guidelines and recommendations for the implementation of software visualization tools. We hope these lessons can ultimately be used profitably by researchers in this field in case they want to start a new implementation or enhance an existing implementation of a software visualization tool.

CodeCrawler's implementation changed and evolved in order to cope with the new requirements our research generated. Its current design is thus able to solve most of the problems we encountered with static software visualizations. Moreover, several of the lessons learned with CodeCrawler can be generalized into more common design guidelines and recommendations which apply to other kinds of software visualization tools as well.

We identify five key issues pertinent to the implementation of a software visualization tool:

1. **the overall architecture**, *i.e.*, the way the software visualization tool as a whole is structured. A clear separation of its three main subsystems, *e.g.*, the core, the visualization engine, and the metamodel, provides for the higher flexibility that is necessary to be resistant against software evolution processes.

2. **the internal architecture**, *i.e.*, the design of the core domain model. Although simple at first sight, the domain model must be designed for extensibility, since the added and new requirements in terms of the functionality needed by the users have an impact on it.

3. **the visualization engine**, *i.e.*, the way information is visualized. Since software visualization tools have special needs, an off-the-shelf visualization library does not offer the degree of freedom needed by the software visualization tool provider. On the other hand writing a complete visualization library from scratch is a cumbersome and lengthy process that should not burden the software visualization tool provider. We describe a compromise solution that largely satisfied our needs.

4. **the metamodel**, *i.e.*, the way data is collected and stored. This part, not directly related to software visualization, but to more general and common reverse engineering issues, should also be separated from a software visualization tool and be reused as an external and well-defined source of functionalities.

[1]This chapter is an extended version of the article *CodeCrawler - Lessons Learned in Building a Software Visualization Tool*, published in the CSMR 2003 proceedings (7th European Conference on Software Maintenance and Reengineering), pp. 409 - 418, IEEE Press, 2003.

5. **the interactive facilities**, *i.e.,* the direct-manipulation possibilities that are offered to the user. Although hard to validate, it is this aspect that requires the most work from a software visualization tool provider and that ultimately dictates the tool's usability and success.

Software and Information Visualization

Price *et. al* [Sta 98] define software visualization as *the use of crafts of typography, graphics design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software*. Ware [WARE 00] states that *visualization provides an ability to comprehend huge amounts of data*. However, software visualizations are often too simplistic and lack visual cues for the viewer to correctly interpret them. In other cases the obtained visualizations are still too complex to be of any real value to the viewer. The goal of any software visualization tool is ultimately to visually render software, be it in a dynamic or static fashion. Software visualization is useful because visual displays allow the human brain to study multiple aspects of complex problems in parallel. All software visualization tools face problems coming form the more general fields of *information visualization* [WARE 00] and *semiotics of graphics (the study of symbols and how they convey meaning)*, wonderfully discussed by Tufte [TUFT 90, TUFT 01, TUFT 97] and Bertin [BERT 74].

Ware [WARE 00] describes four basic stages in the process of data visualization, and interestingly enough these four stages have a direct mapping on the architecture of software visualization tools:

1. The collection and storage of data itself.
2. The preprocessing designed to transform the data into something we can understand.
3. The display hardware and the graphics algorithms that produce an image on the screen.
4. The human perceptual and cognitive system, *i.e.,* the perceiver.

We deduce from these four stages four components that *de facto* must be present in one way or another in every software visualization tool:

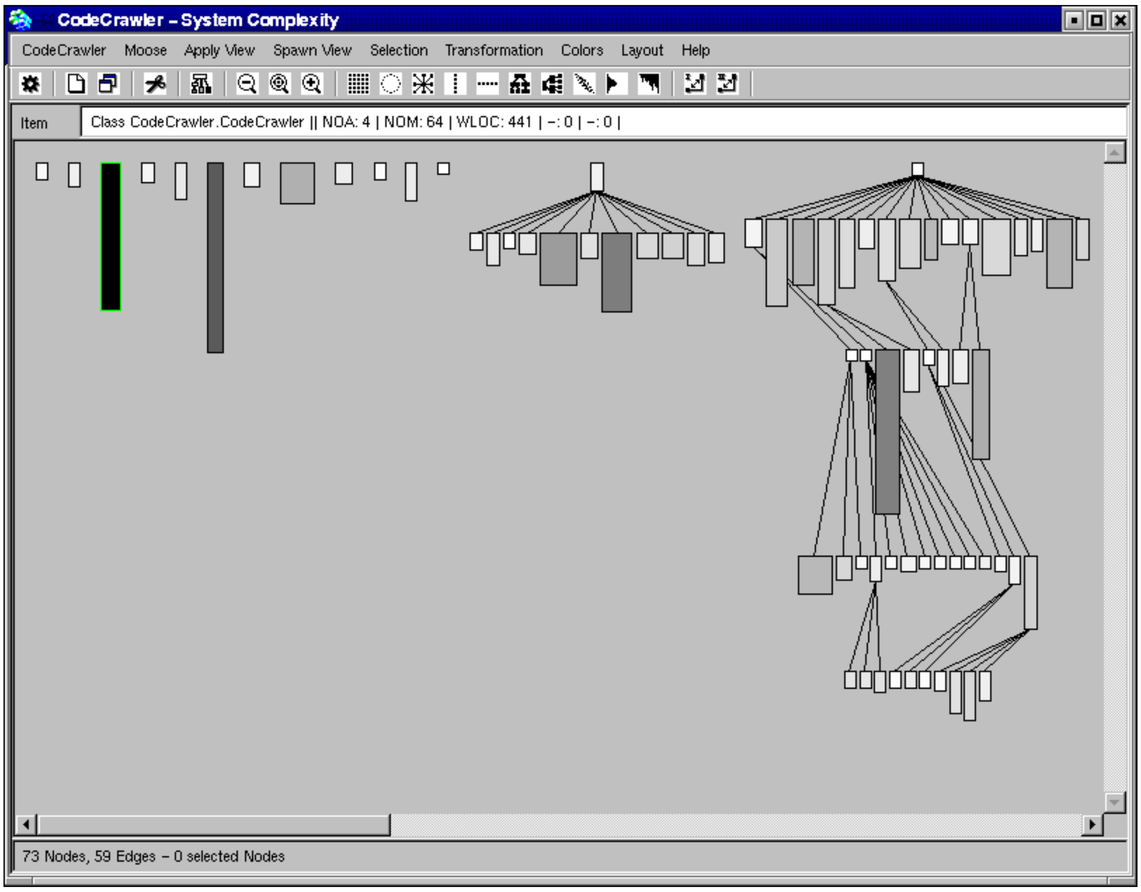
1. **The metamodel**. The data to be visualized, in this case it is software source code, must be collected and stored using a metamodel that provides facilities like parsing, storing, etc.
2. **The internal architecture.** Based on the data provided by the metamodel, a software visualization tool must have some kind of internal representation of what it visualizes.
3. **The visualization engine.** An important part of every tool is devoted to the graphical output of information.
4. **Interactivity.** The perceiver, *e.g.,* the viewer, not only wants to look at software, most of the times he also wants to interact with the visualizations, since static visualizations seldom offer exhaustive explanations to the viewer.

Furthermore the union and interplay of these components can be regarded as the **overall architecture** of a software visualization tool.

Structure of the chapter. In Section A.2 we present our tool CodeCrawler according to the five criteria identified above. We then generalize our findings into lessons learned (Section A.3) before concluding in Section A.4.

A.2 CodeCrawler

CodeCrawler is a language-independent software visualization tool written in Smalltalk. CodeCrawler supports reverse engineering through the combination of metrics and software visualization [LANZ 99, DEME 99b, DUCA 01a, LANZ 01a]. In Figure A.1 we can see a screenshot of CodeCrawler. Its power and


CodeCrawler – System Complexity
CodeCrawler Moose Apply View Spawn View Selection Transformation Colors Layout Help
Item

Class CodeCrawler.CodeCrawler || NOA: 4 | NOM: 64 | WLOC: 441 | –: 0 | –: 0 |

73 Nodes, 59 Edges – 0 selected Nodes

Figure A.1: A snapshot of CodeCrawler's main window. The visualized system in this case is CodeCrawler itself.

flexibility, based on simplicity and scalability, has been repeatedly proven in several large scale industrial case studies. To model software, CodeCrawler uses Moose, a language independent reengineering environment. The first implementation of CodeCrawler started in 1998 as part of a master thesis [LANZ 99]. At the beginning CodeCrawler was based directly on the Smalltalk language, since its reflective capabilities provide for many functionalities that otherwise must be provided by an external metamodel. In 1998 in the context of the FAMOOS ESPRIT project, Moose, the first implementation of the FAMIX metamodel, neared completion and CodeCrawler started to use Moose as metamodel. While FAMIX provided for possibilities as language independence (Java, C++, Smalltalk, Ada, COBOL, etc.) it also involved more complexity. CodeCrawler has had 5 major releases since then, it was last released in October 2002. In its newest re-implementation, described in this chapter, CodeCrawler is now also able to visualize any construct, *e.g.*, constructs and artifacts that do not necessarily come from the field of software reverse engineering.

A.2.1 Overall Architecture

As in every software system, the general architecture of a software visualization tool dictates on one hand how much and which kind of functionality it provides, on the other hand it also defines how it can be extended in case of changing or new requirements.

CodeCrawler adopts what we call a *bridge* architecture described above, as we see in Figure A.2: the internal architecture, *e.g.*, the core of CodeCrawler, acts as a bridge between the visualization engine (on the left) and the metamodel (on the right). It uses as visualization engine the HotDraw framework

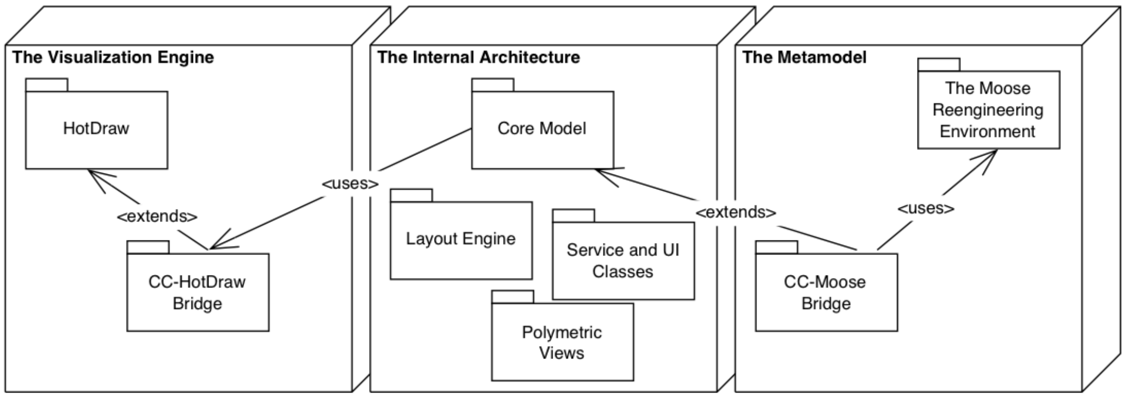


Figure A.2: The general architecture of CodeCrawler, composed of three main subsystems: the core, the metamodel and the visualization engine.

[BRAN 95, JOHN 92] and as metamodel the FAMIX metamodel [DEME 01], whose implementation is called the Moose reengineering environment [DUCA 00] [DUCA 01b]. Both of them are described in more detail later on. In order to keep a certain flexibility CodeCrawler uses facade classes which hide both the visualization engine and the metamodel from the core. It thus can limit the effects of changes happening on the visualization engine and the metamodel. This has the advantage that only the facade classes must be changed when the visualization engine or the metamodel changes. An example of such a change is the newly supported GXL format [HOLT 00], which directly affects only Moose and does not affect the implementation of CodeCrawler, except for adding a new menu item in CodeCrawler's file menu. An example of a major future change is to use a 3D visualization engine. Although at first sight a massive change, this would affect again only the visualization engine's facade classes. In a second moment CodeCrawler would then start to exploit the added third dimension for its visualizations and only then this would have an effect on the implementation of its core.

A.2.2 Internal Architecture

The internal architecture of software visualization tools is largely dictated by their domain model. This depends on the type of visualizations the tool provides. CodeCrawler is focused on visualizing static information about software, *i.e.*, thus working mainly at a structural level. Other visualizations types, not discussed here, include *algorithm visualization and animation*, *computation visualization*. According to the taxonomy presented by Price *et al.*[PRIC 93] CodeCrawler is a *static code visualization* tool.

The internal architecture of CodeCrawler, *i.e.*, all things not related to the visualization engine or the metamodel, can be divided into four parts: (1) the core model, (2) the polymetric views subsystem, (3) the layout engine and (4) the user interface and service classes.

1. **The Core Model.** We can see a simple class diagram of CodeCrawler's core model in Figure A.3. CodeCrawler uses nodes to represent entities (classes, methods, subsystems, etc.) and edges to represent relationships (inheritance, invocation, access, etc.) between the entities. The nodes and edges are contained within a class that represents a graph in the mathematical sense. Both the node class (CCNode) and the edge class (CCEdge) inherit from an abstract superclass which represents a general item (CCItem). CCItem serves as bridge between the visualization part (it contains an attribute named figure which points to a figure class). It is also a bridge to a parallel plugin hierarchy (it contains an attribute named plugin which points to a plugin class). The classes in the plugin hierarchy provide most of the functionality of the nodes and edges. We decided to separate this functionality into an own hierarchy (instead of putting it inside the node and edge classes) in order to obtain more flexibility and a higher degree of extensibility. The plugin hierarchy ultimately serves as another bridge [GAMM 95] to the metamodel representing the software. In our case the abstract superclass

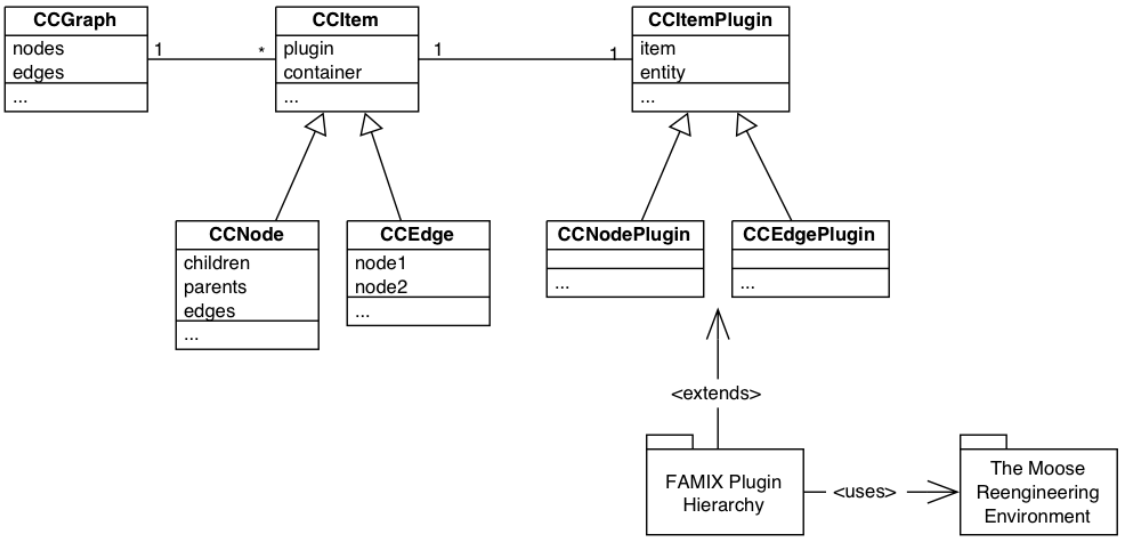


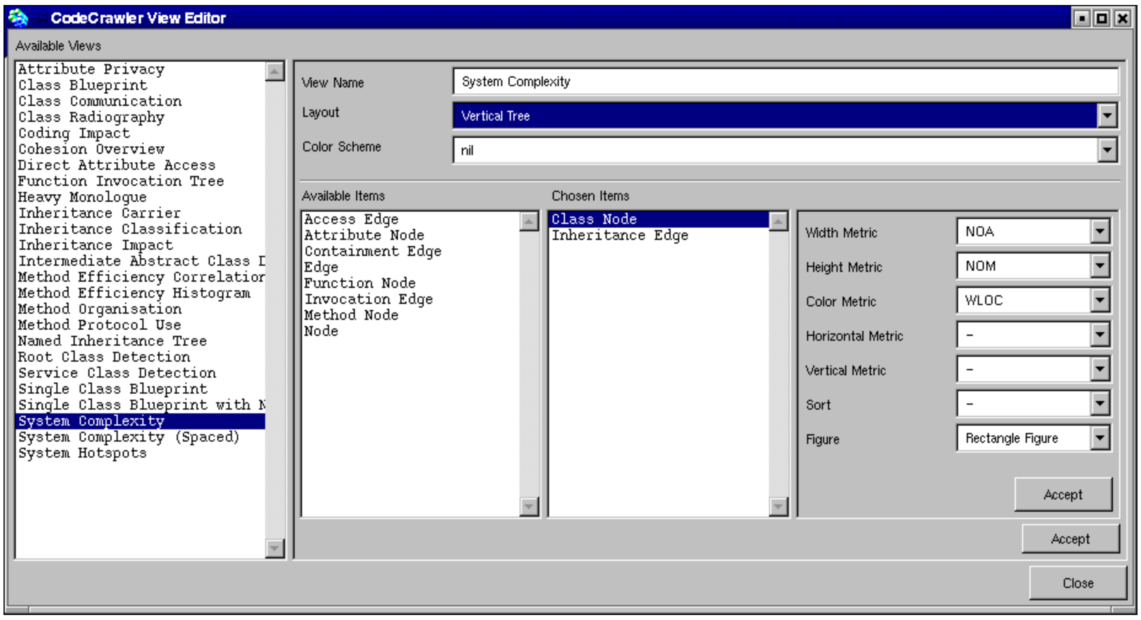
Figure A.3: The core model of CodeCrawler.

CCItemPlugin defines an attribute named *entity* which points to the needed class, *e.g.*, in the case of visualizing software it point to a class in Moose which represents a software artifact. To protect against changes in the metamodel we use again facade classes, *i.e.*, in CodeCrawler we implemented a hierarchy of FAMIX plugins which have counterparts in Moose. To make an example, in order to represent a FAMIX class in Moose (called at this time MSEClass), CodeCrawler implements a CCFAMIXClassPlugin class which interfaces with MSEClass. The return in extensibility of this implementation became obvious when some students extended CodeCrawler's plugin hierarchy in order to model and visualize other kinds of entities, for example for the fields of concept analysis, web site reengineering and prolog rule repositories.

2. **The Polymetric Views.** All information regarding a certain visualization (what is to be visualized, how, where, which metrics, etc.) is stored by means of a view specification class (CCViewSpec). When it comes to display a view of a software system, a view builder (CCViewBuilder) interprets an instance of a specification class and builds the needed visualization. The specifications of the views are easily composed and modified in the view editor window depicted in Figure A.4.

3. **The Layout Engine.** The complex problems that go with graph drawing and graph layouts have been a subject of research for many years [BATT 99]. The layout class hierarchy is part of CodeCrawler, *i.e.*, we do not use any external or commercial graph layout library. The reason for doing so is that in Smalltalk there is no freely available standardized layout library, as is the case for other programming languages like Java or C++. Although an interfacing with libraries written in C would not have been a problem, we decided against that in order to keep as much control as possible. This trade-off between having or delegating control must be carefully evaluated. In CodeCrawler all layouts (at this time ca. 15) inherit from a common abstract superclass (CCAbstractLayout). A layout class takes as input a collection of node figures and assigns a position to each of them.

4. **The Service and UI Classes.** Besides the classes mentioned above, CodeCrawler contains many more classes which provide for various services, for example storing constants and color mappings. Other classes are pure user interface classes (Dialogs, Panels, etc.). Since these classes do not have any features that are particularly important for software visualization tools, we omit their discussion.


CodeCrawler View Editor

Available Views

```
Attribute Privacy
Class Blueprint
Class Communication
Class Radiography
Coding Impact
Cohesion Overview
Direct Attribute Access
Function Invocation Tree
Heavy Monologue
Inheritance Carrier
Inheritance Classification
Inheritance Impact
Intermediate Abstract Class D
Method Efficiency Correlation
Method Efficiency Histogram
Method Organisation
Method Protocol Use
Named Inheritance Tree
Root Class Detection
Service Class Detection
Single Class Blueprint
Single Class Blueprint with N
System Complexity
System Complexity (Spaced)
System Hotspots
```

View Name	System Complexity
Layout	Vertical Tree
Color Scheme	nil

Available Items

```
Access Edge
Attribute Node
Containment Edge
Edge
Function Node
Invocation Edge
Method Node
Node
```

Chosen Items

```
Class Node
Inheritance Edge
```

Width Metric	NOA
Height Metric	NOM
Color Metric	WLOC
Horizontal Metric	-
Vertical Metric	-
Sort	-
Figure	Rectangle Figure

Figure A.4: CodeCrawler's View Editor. The views are composed piece by piece and can then be directly invoked from the main window.

A.2.3 Visualization Engine

The primary task of a software visualization tool's visualization engine is to provide for the graphical output on the screen.

We can see a simple class diagram of the visualization engine in Figure A.5. CodeCrawler uses as visualization engine the HotDraw framework, a lightweight 2D editor written in Smalltalk, consisting of ca. 150 classes. It provides for basic graphical functionalities like zooming, scaling, elision and comes with a collection of simple figures (rectangles, lines, ellipses, composite figures, etc.) that can be easily reused and extended through subclassing, as CodeCrawler does indeed: the subclasses include CCDrawing, which represents the drawing surface on which the visualization is displayed, and several figures classes (CCRectangleFigure, CCLineFigure, etc.) which add functionality to the quite simple HotDraw figure classes. However, these subclasses do not offer protection against changes in HotDraw, since the subclasses would be affected too. Therefore in CodeCrawler three classes (CCItemFigureModel, CCNodeFigureModel and CCEdgeFigureModel), organized in a small hierarchy, serve as facade classes for the figure classes that subclass HotDraw's classes. This allows us to replace on-the-fly the graphical representation, *e.g.*, the figure, of a node or an edge. Furthermore, the facade classes implement several operations that we want to effect on figures (graphical operations, geometric transformations, etc.) and delegate them to the appropriate concrete figures on the screen.

A.2.4 Metamodel

The primary task of a metamodel is to collect and store the data that later on is visualized.

CodeCrawler uses Moose, a language independent reengineering environment written in Smalltalk, to model software systems [DUCA 00, DUCA 01b]. In Figure A.6 we see the internal organization of the Moose reengineering environment. Moose is based on the FAMIX metamodel specification [DEME 01] [TICH 01], which provides for a language independent representation of object-oriented source code and contains the required information for reengineering and reverse engineering tasks like navigation, querying, metrics, refactorings, etc. It is *language independent*, because in the context of the FAMOOS ESPRIT project we needed to work with legacy systems written in different implementation languages. It is *ex-*

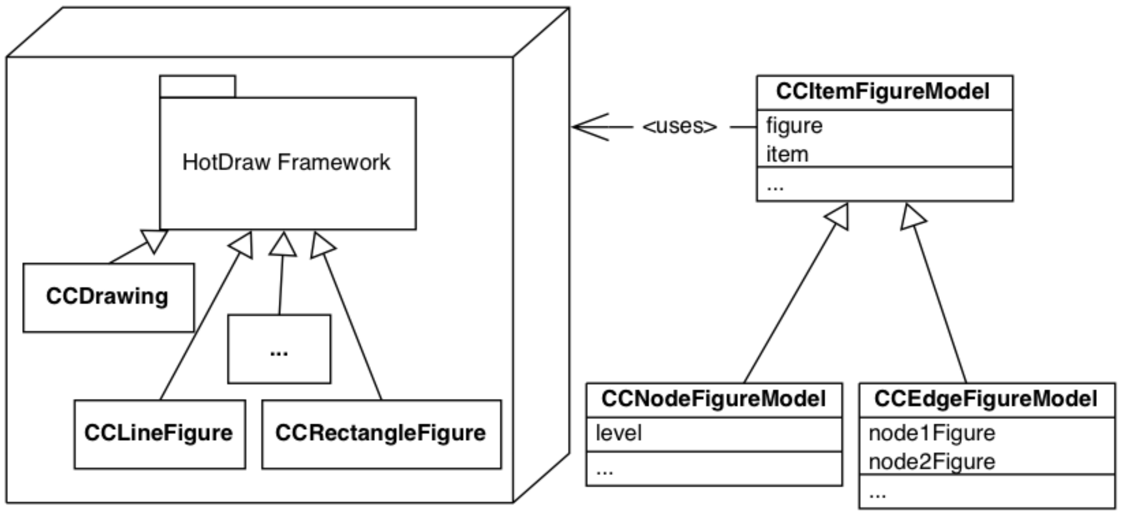


Figure A.5: The visualization engine of CodeCrawler. CodeCrawler subclasses and extends some basic HotDraw figure classes. The class hierarchy composed of the classes CCItemFigureModel, CCNodeFigureModel and CCEdgeFigureModel serves as Facade for the HotDraw figure classes.

tensible, since we cannot know in advance all information that is needed in future tools. Since for some reengineering problems (*e.g.*, refactorings [TICH 01]) the tools might need for language specific information, we allow for language plug-ins that extend the model with language specific features. Next to that we also allow the tool plug-ins to extend the model with tool specific information.

The FAMIX metamodel comprises the main object-oriented concepts – Class, Method, Attribute and Inheritance – plus the necessary associations between them – Invocation and Access (see Figure A.7). Note that the complete FAMIX metamodel includes many more aspects of the object-oriented paradigm, and contains source code entities like formal parameters, local variables, functions, etc. We opted against the use of UML because it is not sufficient for modeling source code for the purpose of reengineering, since it is specifically targeted towards OOAD and not at representing source code as such [DEME 99c].

Moose, a full-fledged reengineering environment, provides CodeCrawler with several services from parsing (Smalltalk and Java) to reading exchange files in different formats (XMI, CDIF, RSF). CodeCrawler uses the functionalities provided by Moose either directly using delegation or by subclassing some of Moose' classes. Furthermore, the plugin hierarchy in CodeCrawler contains a subtree composed of FAMIX plugins, which serves as facade for the actual FAMIX classes in Moose. This is described in detail in the Section A.2.2.

A.2.5 Interactive Facilities

> "Experiments on how people solve spatial problems have uncovered a well-stocked mental toolbox of graphic operations, such as zooming, shrinking, panning, scanning, tracing, and coloring." [PINK 97]

Once the visualization is rendered on the screen, the user not only wants to look at it, he also wants to interact with it. According to Storey *et al.* [STOR 99] this helps to reduce the cognitive overhead of any visualization.

In Figure A.8 we see CodeCrawler at work. In CodeCrawler the HotDraw framework provides for direct manipulation at a purely graphical level, *i.e.*, the user can click, drag, double-click, delete, zoom out/in, spawn child windows, etc. CodeCrawler uses that functionality by providing context-based (pop-up) menus for each node and edge. Note that depending on the type of the node (class, method, etc.)

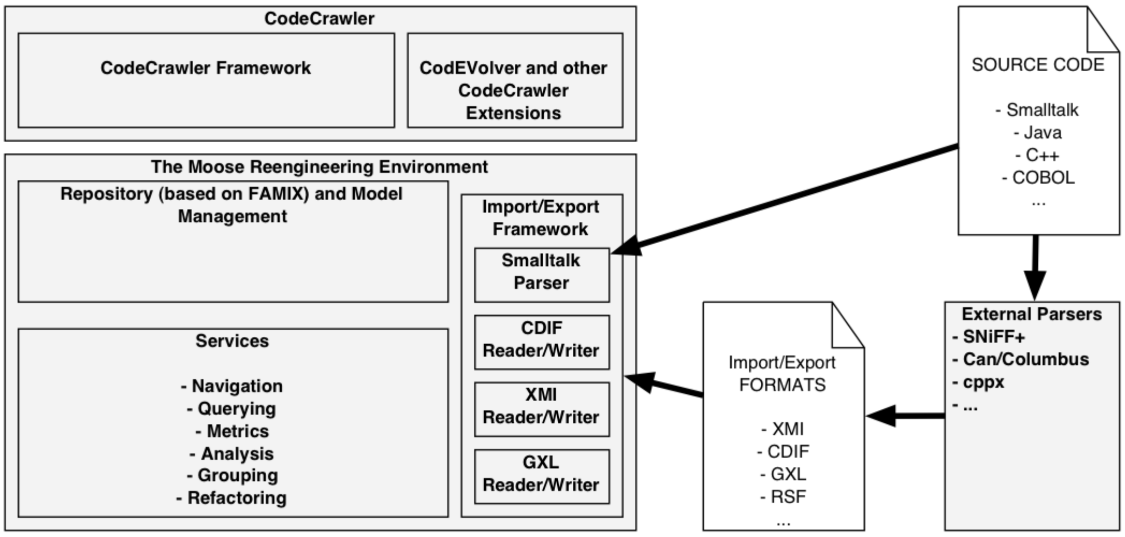


Figure A.6: The architecture of Moose, CodeCrawler's metamodel.

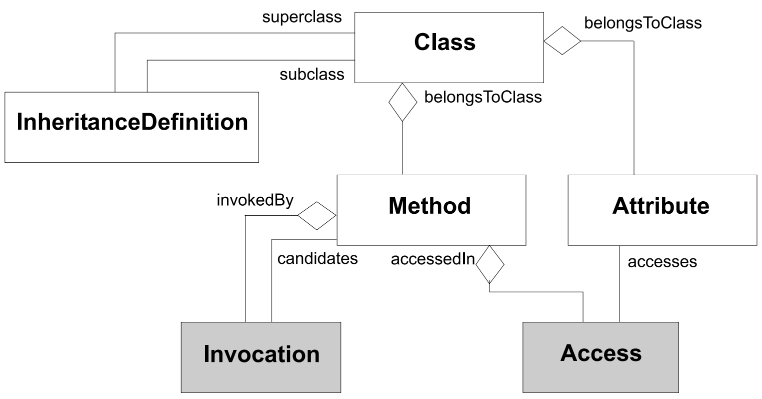


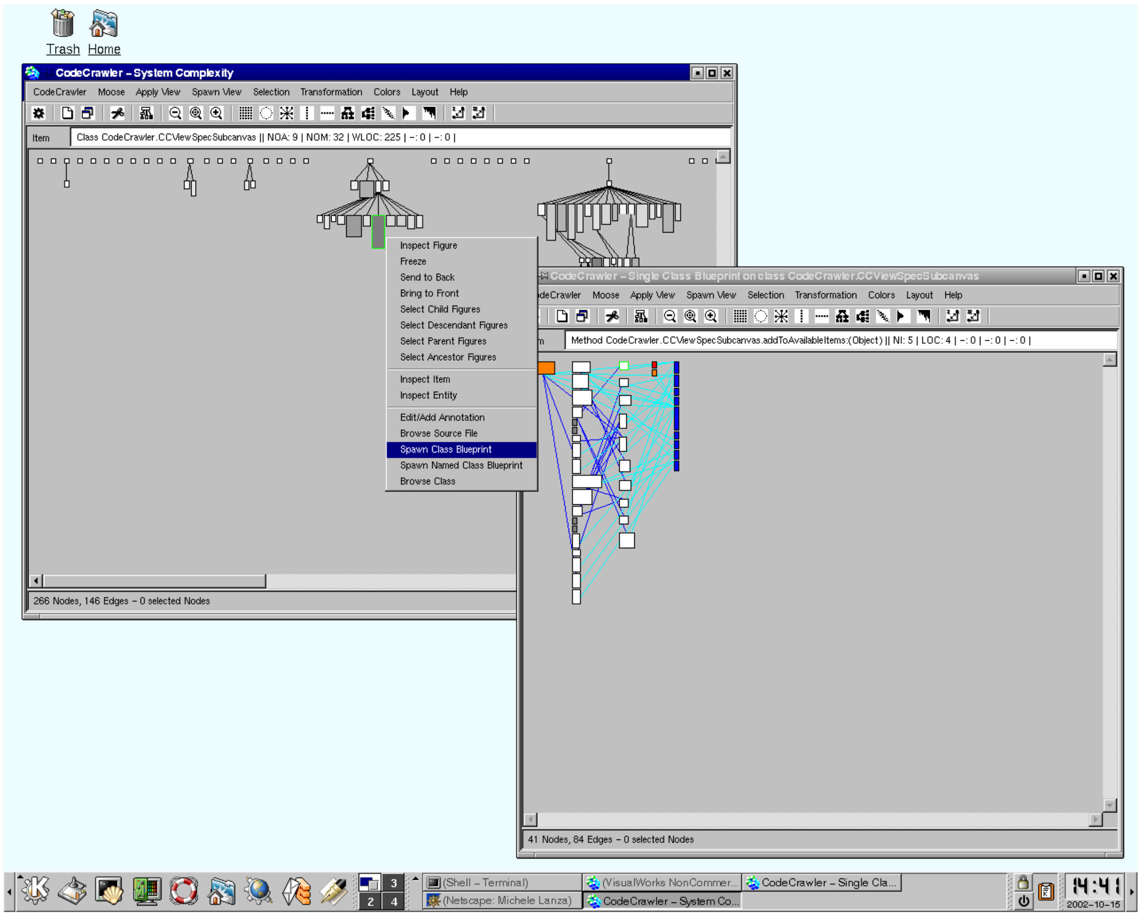
Figure A.7: The core of the FAMIX metamodel.

different choices are offered to the user. For example it is possible to open a class browser on a node, or look at a list of senders of a certain method, etc. In the context of a master in our group [SCHW 02], a student has implemented on top of CodeCrawler several navigation facilities that enable the user to go back/forth from one view to another (macro navigation) or that offer the user context-based navigation aids (micro navigation).

The context menus and the micro navigation are located within the plugin hierarchy, since they are context- or entity-based. The macro navigation and all other graphical interactions like geometric transformations and all multi-windowing techniques are located in CodeCrawler's main window.

A.3 Lessons Learned

In this section we take the lessons learned from the implementation of CodeCrawler and generalize them into more common design guidelines and recommendations. These apply not only for static software visualization tools like CodeCrawler, but in a wider context for visualization tools and reverse engineering tools in general.


Trash
Home
CodeCrawler – System Complexity
CodeCrawler Moose Apply View Spawn View Selection Transformation Colors Layout Help
Item
Inspect Figure
Freeze
Send to Back
Bring to Front
Select Child Figures
Select Descendant Figures
Select Parent Figures
Select Ancestor Figures
Inspect Item
Inspect Entity
Edit/Add Annotation
Browse Source File
Spawn Class Blueprint
Spawn Named Class Blueprint
Browse Class
CodeCrawler – Single Class Blueprint on class CodeCrawler.CCViewSpecSubcanvas
(Shell – Terminal)
(Netscape: Michele Lanza)
(VisualWorks NonCommer...
CodeCrawler – System Co...
CodeCrawler – Single Cla...
14:41
2002-10-15

Class CodeCrawler.CCViewSpecSubcanvas || NOA: 9 | NOM: 32 | WLOC: 225 | –: 0 | –: 0 |

266 Nodes, 146 Edges – 0 selected Nodes

Method CodeCrawler.CCViewSpecSubcanvas.addToAvailableItems:(Object) || NI: 5 | LOC: 4 | –: 0 | –: 0 | –: 0 |

41 Nodes, 84 Edges – 0 selected Nodes

Figure A.8: CodeCrawler at work. The context menus are dynamically built depending on the entity or relationship that is selected.

A.3.1 Overall architecture

In the case of a software visualization tool, as we have seen in Section A.1, the general architecture is readily identified and is composed of (1) the *visualization engine*, (2) the *metamodel*, and (3) the *core* or the *internal architecture*.

1. **The visualization engine.** It provides for the graphical capabilities of the software visualization tool. In some cases the software visualization tool provider uses or extends a commercial or external graphical library, *e.g.*, OpenGL, DirectX, while in other cases he implements it by himself. We do not recommend to implement a graphical library from scratch, as this can become a long and painful implementation marathon without any real improvement of the tool's capabilities. Another design decision that the software visualization tool provider must take is whether he wants to use a 2D or 3D visualization engine. We do not think that 3D involves a much higher complexity, it rather puts more pressure on direct manipulation issues, *i.e.*, how can the visualized software be interacted with and how can it be navigated?

2. **The metamodel.** The metamodel provides for the software visualization tool's data collection and data storage capabilities. The metamodel itself can be language independent (thus providing for a representation of several programming languages at the same time), language dependent or in some

cases even be the language itself without any additional meta-information.

3. **The core.** Ultimately the core is the part of the software visualization tool where the domain model and the tool's functionalities are modeled and implemented.

Both the visualization engine and the metamodel can be considered as *external* tools whose evolution cannot be directly controlled by the software visualization tool provider, unless they provide one or both of them. However, this involves more work which distracts from the implementation of the software visualization tool's core capabilities. It is therefore useful to provide a mechanism of protection against changes happening in either the visualization engine (*e.g.*, the visualization engine is not supported anymore, not up-to-date, does not work on a certain platform, etc.) or in the metamodel (*e.g.*, the implementation changes). By providing the right protection mechanisms it is even thinkable to replace either the visualization engine or the metamodel without having a (big) impact on the software visualization tool core. In our case we do so by means of Facade classes, in a more general case the main point is to define precise interfaces to the both the metamodel and the visualization engine. The quality and stability of these interfaces ultimately defines the overall stability of the tool.

A.3.2 The Internal Architecture

The core task of any visualization tool is to visualize (parts of) this internal graph representation. The visualization can be done by different means, but most tools visualize nodes as rectangles and edges as connecting lines between the rectangles. The internal architecture thus provides for functionalities to allow a visualization. This mainly involves providing guidance to the user and assist him in the process of visualization, *i.e.*, what should be visualized and how?

Many static code visualization tools have adopted as internal representation the basic entity-relationship metamodel, internally represented as a graph consisting of *nodes* (the entities) and *edges* (the relationships).

- **The nodes.** The nodes represent concrete and inconcrete software artifacts. Concrete artifacts can be localized in the source code and include classes, methods, functions, packages, etc., whereas inconcrete artifacts cannot be localized within the source code, but represent often abstractions in the head of the developers. Examples for inconcrete artifacts are groups of classes, subsystems, functionalities, etc.

- **The edges.** The edges represent relationships between the software artifacts. Once again we can identify concrete relationships like inheritance relationships and invocations between methods, and inconcrete relationships between inconcrete artifacts. An example of such a relationship is a dependency between two subsystems ("subsystem A depends on subsystem B") which cannot be localized within the source code.

This representation has the advantage of being domain independent, therefore making a mapping from a domain always possible. However, if domain-related information must be added, the E-R-metamodel is too general. This is where a parallel domain-dependent plugin hierarchy comes into play: we have seen that by using a parallel plugin hierarchy we can separate two concerns: one is the representation of a graph composed of nodes and edges in the mathematical sense, including all operations that go with it (traversing the graph, getting children nodes, etc.), the other is the domain-relevant information, *i.e.*, the node represents a certain software artifact and this information must be modeled as well. One alternative would be to encode everything in the node and edge classes, thus having a deep hierarchy of items, composed of classes like ClassNode, MethodNode, InheritanceEdge, etc. A previous implementation of CodeCrawler adopted this model, but the limits in terms of flexibility soon became evident: the nodes and edges classes became too large, since all item-specific functionality was encoded in them. A separation into two hierarchies practically froze the core model and made it become very stable, while the constant enhancements and additions of functionality have mainly an impact only on the plugin hierarchy.

A.3.3 The Visualization Engine

The visualization engine of a software visualization tool has an influence on its most prominent aspect, the visualization. Indeed, the user perceives it as the tool itself, since he does not see any other internal details. The design decisions to be taken in this case include the (1) type of engine (3D vs. 2D), (2) the degree of possible interactivity, and (3) whether the engine comes from a third party as a possibly commercial product or whether the software visualization tool provider chooses to implement himself the visualization engine as part of the tool.

1. **Engine type (3D vs. 2D).** This decision heavily influences the visualizations provided by the software visualization tool. The use of 3D involves more navigation (*e.g.*, fly-through) and more computing performance. Moreover, the added third dimension must be exploited intelligently, for it is too easy to generate nice looking 3D boxes.
2. **Interactivity.** Direct-manipulation interfaces which allow for several kinds of *direct* interactions are known to be more user-friendly than others. They effectively reduce the latency between the perception of something of interest and the following investigation performed by the user. The user naturally wants to click on the interface to *reduce the distance* between what he sees and what he thinks he sees. This translates to all kinds of interactions, like selecting, moving, removing, copying, inspecting, etc. visualized artifacts. We discuss this aspect in more detail later on.
3. **Implementation.** The decision whether to use or not a third party product (possibly a commercial one) has an impact on the implementation weight that the software visualization tool provider has. Naturally, third party and/or commercial products are more stable, faster and better documented, because the people that provide such products are more experienced and more engaged in graphical issues. It is all too easy for a software visualization tool provider to shift his attention towards the visualization part by providing nicer, faster and more colorful displays at the expense of semantics: Ultimately the goal of a software visualization tool is to provide meaningful information, and not only nice displays, to the user. Therefore reusing graph visualization tools and libraries like Dotty, Grappa and GraphViz can break down the implementation time, but it can also introduce new problems like lack of control, interactivity and customizability.

A visualization engine is merely a vehicle, and not the goal of a software visualization tool. In that sense, although choosing the right engine is important, for the visualization tool the interface to the engine matters. The better-defined it is, the less time the tool provider spends on the engine. In our case we chose to use a lightweight engine which is easily extensible and which provides for all the necessary functionality.

One lesson learned is to choose the appropriate engine and to delegate the job of visualizing as much as possible to the engine. Furthermore, the easier it is to visualize with an engine, the better. In case the tool provider chooses to implement his own engine, we recommend to use a lightweight incremental approach and to strive to obtain visualizations as quickly as possible. Another lesson learned is that keeping as much control as possible over the visualization engine, in terms of implementation, helps to increase the usability of a software visualization tool. The first experiments we did with external engines soon reached a limit, because they were not customizable and flexible enough for our needs. Put in simple words, total (or as much as possible) control is necessary in this case.

A.3.4 The Metamodel

The metamodel used by a software visualization tool has an influence on its internal architecture. In some software visualization tools there is no distinction between the metamodel and the internal architecture. This has the drawback that the tool has a monolithic architecture, and that the concern of the metamodel is not clearly separated from the other parts of the tool. The software visualization tool provider has an interest in making this separation, since the metamodel comes with a considerable level of complexity that should not be added to a software visualization tool's complexity. The reason for this complexity resides in concerns like the collection of information (parsing source code, reading and writing of files in various exchange file formats and the storage and querying of this information (using databases, web browsers, etc.).

The general lesson is to separate metamodel concerns as much as possible from the implementation of a software visualization tool. The software visualization tool must interface to a metamodel and reuse its functionalities, but it should not be tied to it to prevent a mixing of concerns.

A.3.5 The Interactive Facilities

Modern computers allow for faster and more powerful displays, making *direct-manipulation* interfaces possible, which allow the user to not only look at information on the screen, but to interact with it. Several publications and books in the human-computer-interface (HCI) field point out that it is essential to give the user the possibility to ”play” with the displayed information. Since user interface design is not a topic of work we limit ourselves to point out its importance by citing some essential literature by Alan Cooper [COOP 95, COOP 99], Jef Raskin [RASK 00], and Jeff Johnson [JOHN 00].

The interactive facilities a software visualization tool provides heavily influence the quality of the tool in terms of reverse engineering. Storey *et al.* provide a list of 14 cognitive design elements needed for a reverse engineering process [STOR 99]. We deduce from that list that if a tool features direct manipulation it can facilitate navigation by providing directional and arbitrary navigation, while at the same time it reduces disorientation effects by reducing the effort for user-interface adjustment. Put in simple words we can say that the user is more at ease with a tool that supports interactive visualizations.

At the implementation level the problem of interactivity is that it is a cross-cutting concern, *i.e.*, interactivity must be provided by all parts of the system: the visualization engine provides graphical interactivity, while the internal architecture (and the metamodel) provide context-based interactivity. A simple example are pop-up menus, which offer choices at a graphical level (delete figure, spawn window, ...) but also context-based choices (dive into a class node, inspect the senders of a method node, ...). In our case we have seen that the overall bridge architecture is able to cope with this problem: the context menu on a figure is built in succession by the figure, its facade, its item and its plugin, and then presented to the user. Other solutions to this problem is to separate interactive facilities into separate classes and offer them as plugins.

A.4 Conclusion

In this chapter we have presented the internal architecture of a software visualization tool and have identified common problems and issues that are inherent to such systems at various levels. The levels we have discussed include the overall architecture, the internal architecture, the visualization engine, the metamodel and the interactive facilities of software visualization tools.

- **The overall architecture.** An overall architecture which separates the three main parts (core, metamodel, visualization engine) of a software visualization tool allows for higher flexibility and greater extensibility. At the same time the software visualization tool becomes less vulnerable against software evolution processes.

- **The core / internal architecture.** The design of the core of a software visualization tool is largely guided by the goals the tool provider has in mind. Although the notion of a graph consisting of nodes and edges seems trivial, the functionality that matters is the one added to this core and the way this functionality can be used by the tool's users. In the case of CodeCrawler we have seen that there is a separation of the graph notion from a parallel, extensible, plugin hierarchy. This separation allows for a great extensibility through subclassing and addition of functionality.

- **The visualization engine.** The visualization engine's main task is bring the visualizations of software to the screen. However, the degree of integration between a software visualization tool's core and its visualization engine influences the quality of the visualizations. Apart from providing protection mechanisms against changes in the visualization engine, the engine is also largely responsible for the level of interactivity a software visualization tool offers. Seen in this light we do not recommend commercial (black-box) products, but favor white-box products whose classes can easily be reused by delegation or subclassing. In the case of CodeCrawler we protect it against changes by using a facade [GAMM 95] and use and extend a freeware, lightweight visualization framework called Hot-Draw. Note that some visualization engines provide a graph layout library as well. We recommend to use such libraries, because they can greatly reduce the complexity of a software visualization tool.

- **The metamodel.** The metamodel's task is to collect and store the data that is visualized by the software visualization tool. We recommend the separation of the metamodel from the software visualization tool in order to keep the focus on the core functionalities of the software visualization tool. The metamodel can be developed by someone else than the tool provider that has more experience in that area. To make an example, the software visualization tool provider should not have to write a parser by himself, but reuse the existing parsers.

- **The interactive facilities.** Providing interactive facilities to the viewer is essential to the quality of a software visualization tool. While at a purely technical level this should be provided by the visualization engine, the interactions that are enriched with context information are often provided by the the domain model, *i.e.*, the internal architecture of the software visualization tool.

Bibliography

[Car 99] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. Readings in Information Visualization - Using Vision to Think. Morgan Kaufmann, 1999. (pp 11, 54)

[Duc 99] S. Ducasse and S. Demeyer, editors. The FAMOOS Object-Oriented Reengineering Handbook. University of Bern, October 1999. (pp 13, 57)

[ALPE 98] S. R. Alpert, K. Brown, and B. Woolf. The Design Patterns Smalltalk Companion. Addison Wesley, 1998. (p 32)

[BASI 87] V. Basili and R. Selby. *Comparing the Effectiveness of Software Testing Strategies*. IEEE Transactions on Software Engineering, vol. 12, no. 12, pages 1278–1296, December 1987. (p 57)

[BASI 97] V. Basili. *Evolving and Packaging Reading Technologies*. Journal Systems and Software, vol. 38, no. 1, pages 3–12, 1997. (p 57)

[BATT 99] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tolls. Graph Drawing - Algorithms for the visualization of graphs. Prentice-Hall, 1999. (pp 18, 116)

[BECK 97] K. Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1997. (p 46)

[BECK 00] K. Beck. Extreme Programming Explained: Embrace Change. Addison Wesley, 2000. (p 3)

[BERT 74] J. Bertin. Graphische Semiologie. Walter de Gruyter, 1974. (pp 11, 21, 59, 62, 93, 113)

[BRAN 95] J. Brant. HotDraw. Master's thesis, University of Illinois at Urbana-Chanpaign, 1995. (p 115)

[BROW 91] M. H. Brown. *ZEUS: A System for Algorithm Animation and Multi-view Editing*. In Proceedings of the 1991 IEEE Workshop on Visual Languages, pages 4–9, October 1991. (p 54)

[BURD 99] E. Burd and M. Munro. *An Initial Approach towards Measuring and Characterizing Software Evolution*. In Proceedings of the Working Conference on Reverse Engineering, WCRE'99, pages 168–174, 1999. (p 103)

[CASA 97] E. Casais and A. Taivalsaari. *Object-Oriented Software Evolution and Re-engineering (Special Issue)*. Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 233–301, 1997. (p 57)

[CASA 98] E. Casais. *Re-Engineering Object-Oriented Legacy Systems*. Journal of Object-Oriented Programming, vol. 10, no. 8, pages 45–52, January 1998. (pp 1, 7, 10)

[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 17, 21)

[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 1, 8, 10, 13)

[COCK 01] A. Cockburn. Agile Software Development. Addison Wesley, 2001. (p 3)

[CONS 93] M. P. Consens and A. O. Mendelzon. *Hy+: A Hygraph-based Query and Visualisation System*. In Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2, pages 511–516, 1993. (pp 54, 92)

[COOP 95] A. Cooper. About Face - The Essentials of User Interface Design. Hungry Minds, 1995. (pp 56, 123)

[COOP 99] A. Cooper. The Inmates are running the Asylum. SAMS, 1999. (p 123)

[CORB 89] T. Corbi. *Program Understanding: Challenge for the 1990s*. IBM Systems Journal, vol. 28, no. 2, pages 294–306, 1989. (p 57)

[CROS 98] J. H. Cross II, S. Maghsoodloo, and D. Hendrix. *Control Structure Diagrams: Overview and Evaluation*. Journal of Empirical Software Engineering, vol. 3, no. 2, pages 131–158, 1998. (pp 59, 92)

[DAVI 95] A. M. Davis. 201 Principles of Software Development. McGraw-Hill, 1995. (pp 1, 7, 57)

[DEKE 02] U. Dekel. *Applications of Concept Lattices to Code Inspection and Review*. Research report, Department of Computer Science, Technion, 2002. (pp 9, 57, 59)

[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. (p 54)

[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, Proceedings WCRE'99 (6th Working Conference on Reverse Engineering). IEEE, October 1999. (p 113)

[DEME 99c] 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), volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag. (p 118)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 – The FAMOOS Information Exchange Model*. Research report, University of Bern, 2001. (pp 13, 58, 61, 115, 117)

[DEME 02] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. (pp 1, 8, 9, 13, 23, 77, 89)

[DUCA 99] S. Ducasse, M. Rieger, and S. Demeyer. *A Language Independent Approach for Detecting Duplicated Code*. In H. Yang and L. White, editors, Proceedings ICSM'99 (International Conference on Software Maintenance), pages 109–118. IEEE, September 1999. (p 25)

[DUCA 00] S. Ducasse, M. Lanza, and S. Tichelaar. *Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems*. In Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000), June 2000. (pp 13, 95, 115, 117)

[DUCA 01a] S. Ducasse and M. Lanza. *Towards a Methodology for the Understanding of Object-Oriented Systems*. Technique et science informatiques, vol. 20, no. 4, pages 539–566, 2001. (pp 25, 113)

[DUCA 01b] S. Ducasse, M. Lanza, and S. Tichelaar. *The Moose Reengineering Environment*. Smalltalk Chronicles, August 2001. (pp 115, 117)

[EICK 92] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. *SeeSoft—A Tool for Visualizing Line Oriented Software Statistics*. IEEE Transactions on Software Engineering, vol. 18, no. 11, pages 957–968, November 1992. (pp 54, 92)

[FAVR 01] J.-M. Favre. *GSEE: a Generic Software Exploration Environment*. In Proceedings of the 9th International Workshop on Program Comprehension, pages 233–244. IEEE, Mai 2001. (p 54)

[FENT 96] N. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, Second edition, 1996. (pp 12, 16, 17, 54)

[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 54)

[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 54)

[FOWL 99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999. (pp 46, 100)

[GALL 97] H. Gall, M. Jazayeri, R. R. Klösch, and G. Trausmuth. *Software Evolution Observations Based on Product Release History*. In Proceedings of the International Conference on Software Maintenance 1997 (ICSM'97), pages 160–166, 1997. (p 103)

[GALL 98] H. Gall, K. Hajek, and M. Jazayeri. *Detection of Logical Coupling Based on Product Release History*. In Proceedings of the International Conference on Software Maintenance 1998 (ICSM'98), pages 190–198, 1998. (p 103)

[GAMM 95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns. Addison Wesley, Reading, Mass., 1995. (pp 2, 29, 30, 32, 39, 58, 59, 66, 67, 76, 88, 98, 115, 124)

[HEND 96] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1996. (pp 12, 16, 17, 54)

[HEND 02] D. Hendrix, J. H. Cross II, and S. Maghsoodloo. *The Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities*. IEEE Transactions on Software Engineering, vol. 28, no. 5, pages 463 – 477, may 2002. (pp 9, 57, 59, 92)

[HIGG 87] D. A. Higgins and N. Zvegintzov. Data Structured Software Maintenance: The Warnier/Orr Approach. Dorset House, January 1987. (p 92)

[HOLT 96] R. C. Holt and J. Pak. *GASE: Visualizing Software Evolution-in-the-Large*. In Proceedings of WCRE'96, pages 163–167, 1996. (p 102)

[HOLT 00] R. C. Holt, A. Winter, and A. Schürr. *GXL: Towards a Standard Exchange Format*. In Proceedings WCRE'00, November 2000. (p 115)

[INGA 97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. *Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself*. In Proceedings OOPSLA'97, pages 318–326, November 1997. (p 59)

[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. (pp 10, 103)

[JERD 97] D. J. Jerding, J. T. Stansko, and T. Ball. *Visualizing Interactions in Program Executions*. In Proceedings of ICSE'97, pages 360–370, 1997. (pp 9, 54, 92)

[JOHN 92] R. E. Johnson. *Documenting Frameworks using Patterns*. In Proceedings OOPSLA '92, pages 63–76, October 1992. (p 115)

[JOHN 00] J. Johnson. GUI Bloopers. Morgan Kaufmann, 2000. (p 123)

[KAZM 95] R. Kazman and M. Burth. *Assessing Architectural Complexity*. Research report, University of Waterloo, 1995. (p 54)

[KAZM 99] R. Kazman and S. J. Carriere. *Playing detective: Reconstructing software architecture from available evidence.* Automated Software Engineering, April 1999. (p 92)

[KLEY 88] M. F. Kleyn and P. C. Gingrich. *GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views*. In Proceedings OOPSLA '88, pages 191–205, November 1988. (pp 54, 57, 59, 92)

[KLIM 96] E. J. Klimas, S. Skublics, and D. A. Thomas. Smalltalk with Style. Prentice-Hall, 1996. (pp 47, 82)

[KONI 01] G. G. Koni-N'sapu. A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems. Diploma thesis, University of Bern, June 2001. (p 101)

[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. (pp 54, 57, 59, 67, 92)

[LANZ 99] M. Lanza. Combining Metrics and Graphs for Object Oriented Reverse Engineering. Diploma thesis, University of Bern, October 1999. (pp 16, 18, 25, 113, 114)

[LANZ 01a] M. Lanza and S. Ducasse. *A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint*. In Proceedings of OOPSLA 2001, pages 300–311, 2001. (pp 94, 113)

[LANZ 01b] M. Lanza. *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution), pages 37 – 42, 2001. (p 95)

[LANZ 03] M. Lanza. *CodeCrawler - Lessons Learned in Building a Software Visualization Tool*. In Proceedings of CSMR 2003, page to be published. IEEE Press, 2003. (p 3)

[LEHM 85] M. M. Lehman and L. Belady. Program Evolution - Processes of Software Change. London Academic Press, 1985. (pp 3, 7)

[LEWE 98] C. Lewerentz and F. Simon. *A Product Metrics Tool Integrated into a Software Development Environment*. In Object-Oriented Technology Ecoop'98 Workshop Reader, volume 1543 of *LNCS*, pages 256–257, 1998. (p 54)

[LIEB 89] K. J. Lieberherr and A. J. Riel. *Contributions to Teaching Object Oriented Design and Programming*. In Proceedings OOPSLA '89, ACM SIGPLAN Notices, pages 11–22, October 1989. (p 77)

[LITT 96] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. *Mental Models and Software Maintenance*. In Soloway and Iyengar, editors, Empirical Studies of Programmers, First Workshop, pages 80–98, 1996. (pp 4, 58, 92)

[LORE 94] M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice-Hall, 1994. (pp 12, 16, 54)

[LOUN 98] H. Lounis, H. A. Sahraoui, and W. L. Melo. *Vers un modèle de prédiction de la qualité du logiciel pour les systèmes à objets*. L'Objet, Numéro spécial Métrologie et Objets, vol. 4, no. 4, December 1998.

[M.-A 01] C. B. M.-A. D. Storey and J. Michaud. *SHriMP Views: An Interactive and Customizable Environment for Software Exploration*. In Proceedings of International Workshop on Program Comprehension (IWPC '2001), 2001. (p 12)

[MAYR 96] J. Mayrand, C. Leblanc, and E. M. 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 54)

[MEND 95] A. Mendelzon and J. Sametinger. *Reverse Engineering by Visualizing and Querying*. Software - Concepts and Tools, vol. 16, pages 170–182, 1995. (pp 57, 59, 92)

[MÜ 86] H. A. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986. (pp 3, 9, 54, 56)

[NASS 73] I. Nassi and B. Shneiderman. *Flowchart Techniques for Structured Programming*. SIGPLAN Notices, vol. 8, no. 8, August 1973. (p 92)

[PARN 94] D. L. Parnas. *Software Aging*. In Proceedings of International Conference on Software Engineering, 1994. (pp 1, 7, 9)

[PAUW 93] W. D. Pauw, R. Helm, D. Kimelman, and J. Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proceedings OOPSLA '93, pages 326–337, October 1993. (pp 54, 92)

[PAUW 99] W. D. Pauw and G. Sevitsky. *Visualizing Reference Patterns for Solving Memory Leaks in Java*. In R. Guerraoui, editor, Proceedings ECOOP'99, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag. (pp 54, 92)

[PETR 95] M. Petre. *Why looking isn't always seeing: Readership skills and graphical programming*. Communications of the ACM, vol. 38, no. 6, pages 33–44, June 1995. (p 58)

[PINK 97] S. Pinker. How the Mind Works. W. W. Norton, 1997. (p 118)

[PRIC 93] B. A. Price, R. M. Baecker, and I. S. Small. *A Principled Taxonomy of Software Visualization*. Journal of Visual Languages and Computing, vol. 4, no. 3, pages 211–266, 1993. (pp 54, 92, 115)

[RAJL 00] V. Rajlich and K. Bennett. *A Staged Model for the Software Life Cycle*. IEEE Computer, vol. 33, no. 7, pages 66 – 71, 2000. (p 8)

[RASK 00] J. Raskin. The Humane Interface. Addison Wesley, 2000. (p 123)

[RAYS 98] D. Rayside, S. Kerr, and K. Kontogiannis. *Change and Adaptive Maintenance Detection in Java Software Systems*. In Proceedings of WCRE'98, pages 10–19. IEEE Computer Society, 1998. (p 102)

[REIS 90] S. P. Reiss. *Interacting with the FIELD environment*. Software - Practice and Experience, vol. 20, pages 89–115, 1990. (pp 54, 92)

[RICH 99] T. Richner and S. Ducasse. *Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information*. In H. Yang and L. White, editors, Proceedings ICSM'99 (International Conference on Software Maintenance), pages 13–22. IEEE, September 1999. (pp 2, 9, 54, 92)

[RICH 02] T. Richner and S. Ducasse. *Using Dynamic Information for the Iterative Recovery of Collaborations and Roles*. In Proceedings of ICSM'2002 (International Conference on Software Maintenance), October 2002. (pp 9, 54)

[RIEL 96] A. J. Riel. Object-Oriented Design Heuristics. Addison Wesley, 1996. (pp 20, 31, 89, 100)

[RIVA 98] C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. Master's thesis, Politecnico di Milano, Milan, 1998. (pp 103, 109)

[ROBE 97] D. Roberts, J. Brant, and R. E. Johnson. *A Refactoring Tool for Smalltalk*. Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997. (p 20)

[RUGA 98] S. Rugaber and J. White. *Restoring a Legacy: Lessons Learned*. IEEE Software, vol. 15, no. 4, pages 28–33, July 1998. (pp 1, 7)

[SAHR 00] H. A. Sahraoui, M. Boukadoum, H. Lounis, and F. Ethève. *Predicting Class Libraries Interface Evolution: an investigation into machine learning approaches*. In Proceedings of 7th Asia-Pacific Software Engineering Conference, 2000. (p 104)

[SCHW 02] D. Schweizer. Navigation in Object-Oriented Reverse Engineering. Diploma thesis, University of Bern, June 2002. (p 119)

[SOMM 00] I. Sommerville. Software Engineering. Addison Wesley, Sixth edition, 2000. (pp 1, 7, 57)

[STAS 90] J. T. Stasko. *TANGO: A Framework and System for Algorithm Animation*. IEEE Computer, vol. 23, no. 9, pages 27–39, September 1990. (p 92)

[STEI 01] L. Steiger. Recovering the Evolution of Object Oriented Software Systems Using a Flexible Query Engine. Diploma thesis, University of Bern, June 2001. (p 101)

[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 102)

[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. (pp 3, 9, 54, 92)

[STOR 97] M.-A. D. Storey, K. Wong, and H. A. Müller. *How Do Program Understanding Tools Affect How Programmers Understand Programs?* In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings Fourth Working Conference on Reverse Engineering, pages 12–21. IEEE Computer Society, 1997. (p 12)

[STOR 98] M.-A. D. Storey. *A Cognitive Framework for Describing and Evaluating Software Exploration Tools*. PhD thesis, Simon Fraser University, dec 1998. (p 12)

[STOR 99] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. *Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration*. Journal of Software Systems, vol. 44, pages 171–185, 1999. (pp 1, 8, 22, 55, 118, 123)

[TICH 01] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001. (pp 13, 117, 118)

[TILL 94] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. *Programmable Reverse Enginnering*. International Journal of Software Engineering and Knowledge Engineering, vol. 4, no. 4, pages 501–520, 1994. (p 92)

[TUFT 90] E. R. Tufte. Envisioning Information. Graphics Press, 1990. (pp 11, 59, 62, 93, 113)

[TUFT 97] E. R. Tufte. Visual Explanations. Graphics Press, 1997. (pp 11, 59, 113)

[TUFT 01] E. R. Tufte. The Visual Display of Quantitative Information. Graphics Press, 2nd edition, 2001. (pp 16, 93, 113)

[VON 96] A. von Mayrhauser and A. Vans. *Identification of Dynamic Comprehension Processes During Large Scale Maintenance*. IEEE Transactions on Software Engineering, vol. 22, no. 6, pages 424–437, June 1996. (pp 4, 58, 92)

[WARE 00] C. Ware. Information Visualization. Morgan Kaufmann, 2000. (pp 15, 54, 59, 93, 113)

[WILD 92] N. Wilde and R. Huitt. *Maintenance Support for Object-Oriented Programs*. IEEE Transactions on Software Engineering, vol. SE-18, no. 12, pages 1038–1044, December 1992. (pp 1, 8, 10, 57, 59, 67)

[WOOL 98] B. Woolf. *Null Object*. In R. Martin, D. Riehle, and F. Buschmann, editors, Pattern Languages of Program Design 3, pages 5–18. Addison Wesley, 1998. (p 91)

[WUYT] R. Wuyts. *StarBrowser*. http://www.iam.unibe.ch/∼wuyts/StarBrowser/. (p 109)

[Sta 98] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. Software Visualization - Programming as a Multimedia Experience. The MIT Press, 1998. (pp 11, 12, 21, 54, 92, 113)

Curriculum Vitae

Personal Information

Name:	Michele Lanza
Nationality:	Italian
Date of Birth:	April 3rd, 1973
Place of Birth:	Avellino, Italy

Education

1999 - 2003:	Ph.D. in Computer Science in the Software Composition Group, University of Bern, Switzerland Subject of the Ph.D. thesis: "Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained and Evolutionary Software Visualization"
1998 - 1999:	Master in Computer Science in the Software Composition Group at the University of Bern, Switzerland Subject of the Master thesis: "Combining Metrics and Graphs for Object Oriented Reverse Engineering"
1993 - 1997:	Student in Computer Science at the University of Bern, Switzerland Minor in Mathematics at the University of Bern, Switzerland Minor in Micro-electronics at the University of Neuchâtel, Switzerland
1989 - 1993:	Linguistic Gymnasium in Aarau, Switzerland
1979 - 1989:	Primary Schools in Italy and Switzerland