Combining Metrics and Graphs for Object Oriented Reverse Engineering

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michele Lanza

1999

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Dr. Stephane Ducasse
Dr. Serge Demeyer

Institut für Informatik und angewandte Mathematik

Further information about this work, the used tools and an *online* version of this document can be found at:
`http://www.iam.unibe.ch/∼scg/`

The address of the author:

Michele Lanza
Feerstrasse 10
CH-5000 Aarau

or

Software Composition Group
Institut für Informatik und angewandte Mathematik - Universität Bern
Neubrückstrasse 10
CH-3012 Bern
Email: `lanza@iam.unibe.ch`
WWW: `http://www.iam.unibe.ch/∼lanza/`

Abstract

A software system may become very large during its evolution, getting less maintainable while its complexity rises. Since replacing the system with a new one is often out of question because of economic considerations, reengineering techniques are being developed to change the system into a form which makes it easier to maintain and to further develop. However, before a system can be reengineered, it has to be reverse engineered in order to understand its nature and inner logic.

This work deals with a lightweight approach to software reverse engineering combining simple graphs with simple object oriented metrics. Our goal is to obtain a simple and scalable graphical display of a system and its parts through which we succeed to visually extract information, which is useful to the understanding of the system and the detection of its design problems.

The primary goal of this work is to put up a repository of combinations of graphs and metrics which are useful to reverse engineer an object oriented system. To validate our approach we implemented a tool called CodeCrawler, which can graphically display source code while providing a layer of interactivity to the user: we use the term *navigating the code*.

We ran CodeCrawler on two Smalltalk case studies and one large industrial case study written in C++. The positive experiences and reactions which we obtained are a proof of the usefulness of our idea.

Acknowledgments

It's strange how much time one uses to write such a small chapter, which is not even related to the rest of this work.

It doesn't matter to you, it matters to me.

Praise, where praise is due.

I'd like to thank the following people for one reason or another:

- My parents, for giving me life and for being here.
- My brother, long gone, still here.
- My grandfather, for telling me three important things: 1. loneliness is the worst disease 2. it's better to be alone than in bad company 3. friends come and go, the family stays.
- The whole Lanza clan, despite all troubles sure as hell the only family I wanted to be born in.
- Stéphane Ducasse, master of objects, defender of the metaclasses, emperor of Smalltalk, and by far the coolest French I've ever met. Your help and support was more than that. Cool times.
- Serge Demeyer, for his precious help. One of the most brilliant people I'm lucky to know and certainly the coolest Belgian in the galaxy. You will be one hell of a professor.
- Oscar Nierstrasz, head of the Software Composition Group, for his excellent lectures and unexpected but very welcome comments on my work.
- All members of the Software Composition Group.
- Daniele Talerico, Marco Lettere, Marcello Nasso, Thomas Rapold and Livio Dainese, for being my friends. Thanks, guys.
- All friends I've met at the University: Calogero Butera, Daniel Frey, Armin Gemperli, Georges Golomingi, Stefan Martig, David Vogel. Without you it would have been much worse, and this is *not* an understatement.
- Other, innumerable friends that brighten my days and shorten my nights. I'd love to list you all here, but the fear of forgetting someone is too great: You know who you are.

Knock me down, I'll just come back running.

Michele Lanza, 1999

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> *"While the benefits of object-oriented technology are widely recognised, the indiscriminate use of object-oriented mechanisms and weaknesses in analysis and design methods are rapidly leading to a new generation of inflexible legacy systems."* [CASA 98]

The ability to reverse engineer object-oriented legacy systems has become a vital matter in today's software industry. Early adopters of the object-oriented programming paradigm are now facing the problem of transforming their object-oriented legacy systems into full-fledged frameworks, hence need to understand the inner workings of their legacy systems and identify potential design anomalies. However, since legacy systems tend to be big –hundreds of thousands lines of poorly documented code are not an exception– there is a definite need for approaches providing a fast overview and focusing on the problematic parts.

Among the various approaches that exist today, two seem very interesting for large scale reverse engineering:

1. *Program visualisation*, often applied because good visual displays allow the human brain to study multiple aspects of complex problems in parallel[1].
   See [CONS 92], [KLEY 88], [LAMP 95], [MÜLL 86], [DEPA 93], [JERD 97] and [SAND 96], [STOR 95], [SUGI 81], [CROS 98], [BALL 96], [JERD 97] to name but a few.

2. *Metrics*, because metrics are known to scale up well.
   See among others [DEME 99], [KONT 97], [LEWE 98], [LORE 94], [MARI 98].

This paper investigates a hybrid reverse engineering approach based on the combination of graph visualisation and metrics. Moreover, we impose ourselves the extra constraint of simplicity:

- The graph layout should be *quite trivial*.

- The extracted metrics should be *simple* to compute.

Indeed, our goal is to identify useful combinations of graphs and metrics that can be easily reproduceable by reverse engineers using some scriptable reengineering toolset

[1] This is often phrased as "One picture conveys a thousand words".

like Rigi [MÜLL 86, STOR 95] or RainCode[2]. Thus, a reverse engineer should be able to customise a reverse engineering tool in a very short amount of time — say a couple of weeks. Afterwards, the whole reverse engineering team should be able to gain back that time by applying the tool in their daily working practices. To summarise, our goal is to identify useful combinations of simple graphs enriched with metric information that reverse engineers can reproduce easily.

We make use of a range of simple metrics which are easy to calculate to filter information and focus attention. We circumvent the use of composite metrics by exploiting the graphical nature of the visualisation tool, to display up to five different measurements in a single visualisation.

Furthermore we mix the two approaches with a layer of interactivity, which can provide a quick and intuitive way to *navigate* through the code. Instead of taking static snapshots for analysis, we support the use of a dynamic and playful approach towards reverse engineering, because we think that with such an approach, a complex structure like a software system can be understood much more intuitively.

1.1 Goals

We set ourselves a set of goals we would try to reach and questions we would like to have answered during the course of this work. Among these are:

- Put up a repository of graphs which are useful for the reverse engineering of software systems[3] We'd like to have a set of graphs, each of which can emphasise one or more aspects of software structures and be useful for reverse engineering.
- Detect which metrics are useful in this context, and what supplemental metrics have to be developed to further enhance this approach.
- Investigate what the benefits and where the limits are for such a lightweight approach.
- Lay the basis for a methodology consisting of graphs, metrics and interaction which can be used to approach a reverse engineering experience.

1.2 Structure Of This Document

The main document contains the following chapters:

- In Chapter 2 we investigate the problems of reverse engineering, and discuss some possible approaches which have been found to alleviate those problems.
- In Chapter 3 we make a few considerations on object oriented metrics and discuss their usefulness in the software development process.

[2] See http://www.raincode.com for additional information.

[3] In computer science a graph is defined as a set of nodes and edges. In this work we use the term graph in a wider and more visual sense: we mean by it a collection of nodes which may be connected with edges although this is not necessary. The nodes can vary in size and color. Each node represents a language independent metamodel entity, which can be a class, a method or an attribute. Each edge represents a relationship, which can be inheritance, invocation or access. In certain chapters (especially Chapter 4 and Chapter 6) of this document we also use the term *graph* to express its graphical representation as a picture on screen or on paper rather than its scientific definition.

- In Chapter 4 we focus on visualisation techniques and graphs in general. We also see how metrics can be incorporated into graphs with our approach, and what kind of properties a *useful graph* should have to help reverse engineering.

- Chapter 5 contains a short overview of CodeCrawler, the tool which we implemented during this work and which realizes our idea of combining metrics and graphs with interaction. We examine what other frameworks have been used for the implementation of CodeCrawler. We include a section containing some comments about the implementational details of CodeCrawler, with a short discussion on its design and on the central classes. We decided to include this for a possible extension of the program.

- In Chapter 6 we put up a repository of useful graphs. Each graph is discussed in detail as well as the case studies on which we applied them, with a closer look on the obtained results. We include a section of graphs and layouts (Section 6.4), as they are used throughout the chapter.

- In Chapter 7 we explain how to use the graphs discussed in Chapter 6. We set up a possible methodology which can be used to approach a system for reverse engineering using our idea.

- Chapter 8 contains a short resumee of the experiences obtained with our approach during a one week workshop with a large industrial case study.

- Chapter 9, the conclusion, is a resumee of the results of this work and a few considerations on the limits and the potential of the discussed approach. We also include some proposals on possible future work in this context.

The appendix of this document contains the following chapters:

- Chapter A is a dictionary of all graphs and layouts implemented in CodeCrawler which were not used or mentioned in Chapter 6.

Chapter 2

Object Oriented Reverse Engineering

> *“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]

2.1 Introduction

Software systems have become more and more complex due to the fact that they are dealing with more and more complex fields. By adapting themselves to the needs of those fields, they have started to rapidly grow in size and complexity. Since rebuilding a system from scratch each time the specifications change would be too expensive, those systems are mainly being maintained and expanded. The expansion becomes harder with time, since design errors are being dragged along the way, making the further evolution of the system very difficult. The development of new programming techniques [BROO 75] which should facilitate the design and evolution of a system (Object Oriented programming languages, fourth generation languages, visual GUI programming) has brought no real relief to this problem, because these techniques could not cope with the order of magnitude increase in complexity we have been facing in the last two decades.

Although low-level languages like C and Pascal have an appealing property, which is their efficiency and compactness, at a certain level of complexity the overview is lost because programmers often have to care about implementation details instead of keeping an eye on a good and clear design: the more low-level a language is, the harder it becomes for the programmer to think in broader terms at a higher abstraction level. However, thinking at higher abstraction levels has become vital in software engineering because of the enormous complexity of current systems.

As these procedural languages could not ease coping with complexity, in the last two decades object oriented languages have become the main force in software development nowadays. Especially C++, Java and Smalltalk are widely used in industry. Object oriented languages have an important advantage: They divide the world in objects which communicate with each other and which possess attributes (properties). This distinction enables developers to work at a higher abstraction level, which is very important when they deal with large and complex systems.

However, the last decade showed that even using the object oriented paradigm, coping with very large software systems is a hard task: Very large software systems can become several millions of lines of code long, with many different people having taken part on its implementation lasting into months or years. Many problems can affect those systems, naming but a few:

- The original developers left and there is nobody who fully understands the original design decisions.
- Missing, sparse or erroneous documentation [BROO 75, CASA 98, WILD 92].
- Obsolete programming tools, platform migrations and outdated hardware makes it hard to find people knowing such techniques or willing to deal with such problems. A good example in this case is the so-called millennium bug, also known as the Y2K - problem, where suddenly a huge number of experts was needed to test software written in languages which are no longer used nowadays.
- Maintenance is often done by less experienced programmers which have to face not only the problem of complexity but also the problem of dealing with code from unknown areas. In fact, experienced programmers which often tend to move on to other projects and areas of interest, take also a great deal of domain-specific knowledge with them which the maintainers sometimes lack.
- Several design errors have made the evolution of the system nearly impossible: small changes can affect large parts of the system.
- There is duplicated code everywhere, which means the programmers used to copy and paste often. Duplicated code can cause code bloat, error propagation (errors are copied as well!) and decrease flexibility (a change has to be done in many places) [BAXT 98, BAKE 92, DUCA 99].

Even with all those points speaking for a reprogramming from scratch of the system, there is one main point speaking against it: *The system is working.* Maintenance of such systems is thus the only possible approach. [WILD 92] states that maintenance, in its widest sense of 'post deployment software support', is likely to continue to represent a very large fraction of total system cost. Rebuilding the system from scratch would mean months or years of development, but with the ongoing technology race such a long delay can mean financial ruin.

2.2 The Problems of Software Industry

Software industry has a somewhat schizophrenic approach to itself: while the academic faction is preaching high level concepts like architectures, engineering patterns and reusable components, what is practised in reality by the industry is of a wholly different nature.

Systems often resemble a *big ball of mud* as stated in [FOOT 97]:

> *"A haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle. Its code shows unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly*

> *all the important information becomes global or duplicated. The overall structure of the system may never have been well defined. If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun them. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.*"

There are several forces which can produce such systems:

- Lack of time. Looming deadlines can drive out any attempt to bring some architecture and design into the system.
- Complexity and lack of experience. The application domains of today's software systems often require domain-specific knowledge which often cannot be provided by the implementors. This experience often comes with time, when the major design decisions have already been taken and it's too late to rebuild the system from scratch. Another phenomenon is that programmers who have gained domain-specific knowledge after years of development, prefer to change domain because of boredom or lack of interest.
- Rapid employee turnover. When experienced programmers leave an organisation they take important knowledge from the organisation's common memory. The so-called fresh blood brought in by new programmers often means months of building up their understanding, during which their productivity remains at a low level.
- Programmer skills. Programmers differ in their levels of skill, as well in expertise, predisposition and temperament. Skilled programmers prefer to move on to new more interesting tasks, leaving the dirty job of maintenance and evolution of systems to the less experienced ones.
- System changes. Successful systems are especially prone to changes desired by the users. These changes often come at a time where the system has already reached considerable complexity. While redesigning the system would be the only right thing to do, the system if patched to satisfy the new requirements.
- Scalability. An often discussed topic is the size of software systems. Often they are developed over time spans of years or decades, by hundreds or thousands of programmers. Such systems cannot be compared to small projects because of the evident supplemental logistic overhead of the large projects.
- Cost. The practice of rapid prototype development has gained popularity over the last few years. The prototypes, originally intended to show "what a system could do", are often shipped out because the management wishes to do so. This involves that systems have to be patched and changed from the beginning. Good architecture, is often put aside because it is considered expensive. This is somehow true: a perfectly designed system, which is however released after the quick-and-dirty designed one from a competitor, often has selling problems, because the market has already been eaten up by the competitor.

Once these systems are up-and-running, it's hard to convince managers that they have to be redesigned. Major changes to those systems can increase their instability and have to be applied very carefully. There are vital systems which have to work 24 hours a day / 7 days a week. If such a system breaks down, this could involve loss of money, time and possibly human life. Although maintenance needs accumulate, and the systems become obsolete with the years, an overhaul could break them.

This implies that changes have to be applied very carefully, and the prerequisite for such changes is clearly a good understanding of the system.

2.3 Software Reengineering

> *"Reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. It generally includes some form of reverse engineering (to achieve a more abstract definition) followed by some form of forward engineering or restructuring. This may include modifications with respect to new requirements not met by the original system."* [CHIK 90].

It's important to understand that without reverse engineering, the reengineering of a system is unthinkable: changing a large and complex system without sufficient knowledge of its inner structure, will almost certainly trigger unwanted side effects which could make the system inoperable.

Software Reengineering can roughly be divided into three steps, the first two of which have been termed as *reverse engineering*. [CHIK 90].

1. **Understanding and Design Extraction**. We have to gain knowledge of the system. This can be achieved in many ways:

    - Reading the manuals if they are present.
    - Talking to one of the original developers.
    - Looking at the source code in a textual or graphical form.
    - Running the software and see how it behaves.

    Once we have understood the inner workings of the system, we should try to distill the original design decisions which made the program become the way it is.

2. **Problem detection**. Once we have extracted the design, we can detect wrong design decisions or areas of the system which should be refactored and redesigned.

3. **Reengineering**. Once we know where the problems lie, the system has to reengineered, i.e. transformed into a state which makes it easier to maintain and to further develop. For this purpose several reengineering tools, techniques and patterns have recently been developed [CASA 98].

2.4 Reverse Engineering

The problem which arises when we try to understand a system, is the huge amount of information we have to keep in mind. Furthermore we have to discern the important pieces of information from the less important. We have to know which are the important classes or subparts of the system. How can this be done? Scrolling through thousands of lines of code and browsing hundreds of classes is of no use, we'd have to spend weeks just to get a feeling for the system. We could try to stick to rules like 'look for the big ones, they're probably important', but experience has shown that such ad hoc rules fail far too often [DEME 99]. There are two main paths that are followed these days:

1. *Metrics*. A first good approach is to use metrics and to *measure* the system. We count the number of methods in a class, the lines of code of a method, the hierarchy nesting level of a class, etc. There is a wide array of possible metrics we can use, and many have been proposed [LORE 94]. Once we have measured the system we can make assessments about it. The problem is, that even with this kind of information we cannot really grasp the inner workings of the system. However, their advantage is that they can enrich the semantic properties of a software entity, which can express its size or complexity with a simple number. We will discuss metrics in depth in Chapter 3.

2. *Program Visualisation*. A second, often used approach is to represent source code graphically. Such a representation introduces an abstraction level which hides the source code behind a graphical entity. Several techniques exist in this domain, which include complex layout algorithms, filtering and interaction. We will further discuss this issue in Chapter 4.

To summarise, reverse engineering is mainly about understanding complex things. As the textual representation of source code hinders such an understanding in case of a large and complex structure, techniques have been developed to alleviate this problem. All techniques have in common that they provide an abstraction of the code while at the same time they must cope with scalability and complexity, i.e. if possible they should work with any amount of code and not break at a certain level of complexity.

Chapter 3

Object Oriented Software Metrics

3.1 Introduction

Improving software product quality and performance and development team productivity has become a primary priority for almost every organisation that relies on computers. The process of developing new software and maintaining old systems has in many cases been poorly implemented, resulting in large cost overruns and squandered business opportunities [MOLL 93]. Indeed, the software problem is huge. The main reason for this is that often estimations on time and cost are based on feelings rather than numbers.

The rise in popularity of object oriented methods raises questions about how we measure object-oriented structures[FENT 97]. A milestone in this regard was the paper *A Metrics Suite for Object Oriented Design* by Shyam R. Chidamber and Chris F. Kemerer, where an attempt was made of a formal definition of metrics based upon measurement theory [CHID 91, CHID 94].

The paper had three main goals:

1. Propose metrics that are constructed with a firm basis in theoretical concepts in measurement and the ontology of objects, and which incorporate the experiences of professional software developers.

2. Evaluate the proposed metrics against established criteria for validity.

3. Present empirical data from commercial projects to illustrate the characteristics of these metrics on real applications, and suggest ways in which these metrics may be used.

Although much discussed and criticised [CHUR 95, HITZ 95] afterwards, it broke new ground because of its clarity and precision and it cleared the way for further research [LORE 94, FENT 97] in this field, which has by now become a recognised discipline of software engineering.

The metrics that have been developed over the years can be divided in two groups. We use the definitions used in [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.

3.2 The Metrics In Our Project

In this project we make extensive use of object oriented software metrics.

However, we don't use every possible metric as we focus ourselves on design metrics for several reasons:

- We make use of metrics in a very concrete way: we use them to assess the size and complexity of software entities. For that reason we can't make use of project metrics, as they can't be used in that regard.

- As the lightweight aspect of our approach is one of its most important properties, we decided to apply this aspect for the used metrics as well: we chose to use metrics that have a simple definition which can be directly expressed. As such we dont make use of *composite metrics* which raise the issue of dimensional consistency: since one cannot compare apples and oranges, care is demanded when combining different measurements [HEND 96]. Furthermore we don't make use of *indirect measurement*, where metrics are combined and calculated. A good example for such metrics are average and percentage metrics.

Our idea is the following: object oriented languages introduced the idea of entities (classes), which have properties (variables, attributes) and which communicate by invoking methods. These three types of entities can not only be identified by unique names, we can also assign numerical values to them. By that we mean that they can be measured in many ways and these measures constitute a considerable part of their identity.

We are convinced that for a reverse engineer the notion of size (or complexity) of an entity can mean much more than a name, which is often domain-specific. For that reason we need metrics which can be termed as *direct measurement* metrics [FENT 97], i.e. they involve no other attribute or entity.

We now list all metrics we used during this project, and which were implemented in the Moose metamodel discussed in Chapter 5. The metrics are divided into three groups, namely class, method and attribute metrics, i.e. these are the entities that the metric measurements are assigned to. The following tables of metrics contain an acronym (name) and its description.

3.2.1 Class Metrics

In Table 3.1 we list every class metric that was used during this project. Classes, which are the central points of every object oriented language implement methods and define attributes. The class metrics address thus this aspect: their complexity can be expressed through methods and attributes[1] and the way these entities behave.

Name	Description
HNL	*Hierarchy nesting level*, also called *depth of inheritance tree*. The number of classes in superclass chain of class. In case of multiple inheritance, count the number of classes in the longest chain.
NA	*Number of accessors*, the number of get/set - methods in a class.
NAM	*Number of abstract methods*.
NC	*Number of constructors*.
NCV	*Number of class variables*.
NIA	*Number of inherited attributes*, the number of attributes defined in all superclasses of the subject class.
NIV	*Number of instance variables*.
NMA	*Number of methods added*, the number of methods defined in the subject class but not in its superclass.
NME	*Number of methods extended*, the number of methods redefined in subject class by invoking the same method on a superclass.
NMI	*Number of methods inherited*, i.e. defined in superclass and inherited unmodified.
NMO	*Number of methods overridden*, i.e. redefined in subject class.
NOC	*Number of immediate children of a class*.
NOM	*Number of methods*, each method counts as 1. NOM = NMA + NME + NMO.
NOMP	*Number of method protocols*. This is Smalltalk - specific: methods can be grouped into method protocols.
PriA	*Number of private attributes*.
PriM	*Number of private methods*.
ProA	*Number of protected attributes*.
ProM	*Number of protected methods*.
PubA	*Number of public attributes*.
PubM	*Number of public methods*.
WLOC	*Lines of code*, sum of all lines of code in all method bodies of the class.
WMSG	*Number of message sends*, sum of number of message sends in all method bodies of class.
WMCX	*Sum of method complexities*.
WNAA	*Number of times all attributes defined in the class are accessed*.
WNI	*Number of method invocations*, i.e. in all method bodies of all methods.
WNMAA	*Number of all accesses on attributes*.
WNOC	*Number of all descendants*, i.e. sum of all direct and indirect children of a class.
WNOS	*Number of statements*, sum of statements in all method bodies of class.

Table 3.1: The class metrics used in this project.

3.2.2 Method Metrics

In Table 3.2 we list every method metric used in this project. Methods can be seen as a flow of instructions which take input through parameters and which produce output. Methods can invoke other methods or access attributes. The method metrics are defined in this context.

[1] In this work the terms instance variable and attribute are used interchangeably. We tend to use the term attribute because it's more general, but in most cases and if not mentioned otherwise we are talking about instance variables.

Name	Description
LOC	*Lines of code* in method body.
MHNL	*Hierarchy nesting level* of class in which method is implemented.
MSG	*Number of message sends* in method body.
NI	*Number of invocations* of other methods in method body.
NMAA	*Number of accesses on attributes* in method body.
NOP	*Number of parameters* which the method takes.
NOS	*Number of statements* in method body.
NTIG	*Number of times invoked by methods non-local to its class*, i.e. from methods implemented in other classes.
NTIL	*Number of times invoked by methods local to its class*, i.e. from methods implemented in the same class.

Table 3.2: The method metrics used in this project.

3.2.3 Attribute Metrics

In Table 3.3 we list every attribute metric used in this project. Attributes as properties to classes. Their main function is to return their value when accessed by methods. The attribute metrics are defined in such a context.

Name	Description
AHNL	*Hierarchy nesting level* of class in which attribute is defined.
NAA	*Number of times accessed.* NAA = NGA + NLA.
NCM	*Number of classes having methods that access it.*
NGA	*Number of times accessed by methods non-local* to its class.
NLA	*Number of times accessed by methods local* to its class.
NM	*Number of methods accessing it.*

Table 3.3: The attribute metrics used in this project.

Chapter 4

Combining Metrics and Graphs

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

4.1 Introduction

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

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

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

4.2 Graphs for Reverse Engineering

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

- *Simplicity and Quality*. The first important prerequisite is that the generated pictures of a graph have to be relatively simple and easy to grasp. The main reason for that is that too much displayed information overloads the viewer's perception. This tends to backfire and causes an unwanted information loss. A secondary aspect is that simple graphs are also easily reproducible, while complex techniques like hyperbolic trees [LAMP 95] are affected by a considerable complexity which is hard to grasp and reproduce. Many approaches have been discussed as to how a software entity could be represented for program visualisation ([BALL 96, DEPA 93, KLEY 88] to name but a few). We think that a graphical representation of an object oriented entity should be easy to grasp and not make use of a specific dictionary of shapes which has first to be learned. A graph should be able to transmit useful information to the viewer at first sight.

- *Quantity*. We have to be able to select how much of the subject system we want to display and at what level of granularity. Thus, we should be able to zoom in and out of such a graph and reduce the amount of displayed information at will.

- *Colors*. Program visualisation can be supported by colors, because they can attract the eye to interesting hot spots, while other parts of the graph which look less colorful can be ignored by the viewer. Colors have often been used in program visualisation [RIVA 98]. While colors are a good way to attract the attention of the eye, the usage of too many colors in a graph is not advised, since this results in an optical overload for the viewer of the graph. We also advise against the use of color conventions which have first to be learned by the viewer, as this lessens the impact of the colors.

- *Scalability*. As reverse engineering is especially crucial in very large systems, a visualisation should be scalable and work if possible at any level of granularity. The number of displayed entities should not affect the quality of the graph.

- *Interactivity*. A very important aspect of graphs is not only their layout algorithm but also that they can provide interactivity to the user through direct-manipulation interfaces. Making a static display of nodes and trying to extract information from the graph has clearly defined limits, which we discuss below in Section 4.2.1.

- *Metrics*. Although intangible in the physical sense, software *has* size. It can be measured, especially in object-oriented code we can assign numerical values (metric measurements) to its entities. Although the concept of software is abstract and often exists only in the head of the programmer, we can measure it. Once we can measure it, we can assign a size to it and represent this size graphically. We think that metrics enrich the semantic value of a graphical representation of a software entity, and discuss this below in Section 4.2.3.

4.2.1 Interactivity

A graph which lacks interactivity has certain drawbacks:

1. The user can't produce new views starting from a part of the graph.

2. The user can't find out secondary information (e.g. he can't inspect the nodes or browse through their source code).

3. The user can't reduce the amount of displayed data by either removing nodes by hand or by filtering out nodes through algorithms.

Those limits can be overridden if the graph is interactive:

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

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

4.2.2 The Use of Layout Algorithms

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

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

However, we don't want to minimise the importance of complex layout algorithms, on the contrary: we believe they could bring many more benefits than drawbacks. Good layout algorithms just were not part of the constraints of this work. But it is certainly a very promising field of research in this context. We go further into the details of this subject in Chapter 9. The layout algorithms used in this work are discussed in detail in Chapter 6 and Chapter A.

4.2.3 The Use of Metrics in Graphs

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

It is obvious that everything regarding metrics possible through their graphical display is also possible by just calculating and analysing the metric measurements. So the question arises why we should have a graphical display of them, since the information sought is in the metrics themselves. But in the same way one could think to listen to music by just reading the partiture of a song instead of using the sense normally designed for that (the hearing)[1]. *What changes is the perception and the impact of what is perceived.*

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

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

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

This concept is rendered clearly in Figure 4.1, where we see where the metrics can be applied on a node.

Not every graph can make use of five the metrics at the same time. In a graph that does not have an origin (which defines an absolute coordinate system) it makes no sense using two metrics for the position of the nodes. A good example for such a graph is a tree graph, where the position on the nodes is implicitly defined by the logical position of the nodes in the tree. Another property which came up during our experiments was that sometimes the multiple use of the same metric (for example if we choose the same metric to reflect width and height) can emphasise certain parts of the graph and render them more clearly for the viewer.

4.2.4 The Node Size Problem

The Problem. We map metric measurements to the size of a rectangle (node). However, we have to make some important considerations regarding the distortion of information.

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

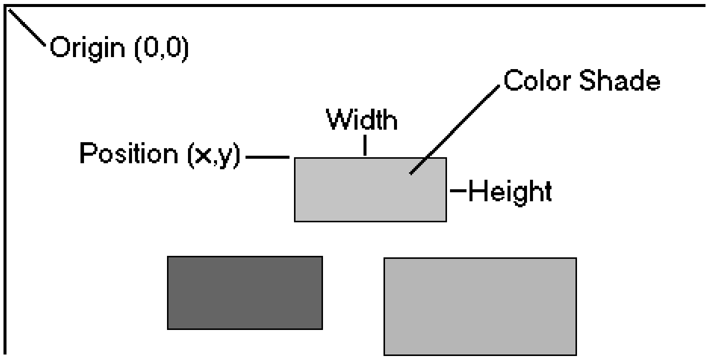


Figure 4.1: An example of nodes and their possible metrics.

If a node has a metric measurement of 5 for its width we want the displayed node on the screen to be 5 pixels wide. But, which width will the node have in case the measurement is zero? The node would not be displayed then because it wouldn't have a size. This is the introduction of the node size problem, which we discuss now.

We have to consider two points:

- Consistency. We have to alleviate the problem of a distorted mapping function. As we have seen in the case of a metric measurement equal to zero, it's not possible to stick to a linear function.
- Interactivity. As we want our graphs to provide direct-manipulation interfaces, we'd like to be able to click with a mouse pointer on a node. This is very hard to impossible for sizes from 1 to 3 pixels.

Possible Solutions. We list here a few solutions which came to our mind:

- Because of the problem above, we decided to introduce a *minimal node size (MNS)*. If the metric measurement was below a certain predefined threshold, we assigned a fixed size to the node.

  Yet this generates another problem: Suppose the chosen minimal node size is 4. We have two different entities whose metric measurement is respectively 1 and 3. With the technique described above both of them are drawn as nodes with a size of 4. However this is a serious distortion of reality: The second entity has a metric measurement which is three times as big as that of the first entity, but nonetheless they are drawn the same size. This cannot be a correct assumption.
- Another solution which we found, was to map all present metrics into a predefined interval which then would be used as node size (we have seen before that this solution works with color shades). The problem is, that this is again an even heavier distortion of reality. In this case the node size would not reflect the metric measurements anymore. So this solution could not be correct either.
- Let us make a few other considerations which come into mind: We could use the MNS as a *size factor*. This would mean that the first node would get a size of 1 * 4 = 4. The second one would get a size of 12 (= 3 * 4). However: what could

we do with a third node whose metric measurement is zero? We could find a workaround by adding 1 to all metric measurements.

Thus the first node would have a size of (1 + 1) * 4 = 8. The second one would have a size of (3 + 1) * 4 = 16. The third one would be (0 + 1) * 4 = 4 pixels big. This approach is not satisfactory either, we still have distorted the values and there is another drawback to this approach: The nodes easily tend to get very big with this approach: Classes often have dozens of methods. The number of nodes displayable on the same screen would rapidly decrease, but we made the use of easily displayable graphs one of the constraints of this work, and we don't want to scroll around through dozens of screens because the drawn graph has gotten so big. This would hinder the intuitive approach to such graphs.

Our Solution. We finally settled for the following compromise: We use the MNS as a starting point for the metric value. This means that if we use a MNS of 4, a node with a metric measurement of 1 would have a size of 4 + 1 = 5 pixels. A node with a metric measurement of 3 would have a size of 7. This is the best compromise we have found, but it is *still a slight distortion of reality*: the second node is **not** three times as big as the first one, although it should be. However, we found this solution more than adequate, as our experiences later showed. Consider also that the smaller the MNS is, the less impact it has on the node size.

Considerations. Note that the same problems are present when it comes to the maximal node size: At first we also tried to restrict the size of nodes to a certain maximum value (because of space intensive graphs which would not fit on one screen), but this approach was even more problematic, since often it's just the big nodes that we are looking for. We opted for the introduction of two shrink factors, one for the width and the other for the height. The user can decide whether he wants to use shrink factors or not. If he opts for that, each time a node size exceeds a certain maximal node size, all nodes are recursively shrunk by factor 2, until the biggest node gets smaller than the defined upper threshold.

This can introduce yet another problem: suppose we are interested in the *shape* of the node. If only the height **or** the width are shrunk, the nodes change their form. The logical solution to this problem was to give the user another choice on whether he wants to shrink the nodes in both dimensions at the same time, even if only one of the dimensions exceeded the threshold.

Conclusion. Let's resume the possible approaches to the node size problem in Table 4.1: We put up three possible metric measurements of 1, 3 and 0 and see how the possible solutions behave. The emphasised column is the solution we chose to adopt in this case.

The Node Size Problem				
Metric Measurement (m)	Resulting size			
	MNS	**m + MNS**	m * MNS	(m + 1) * MNS
1	4	**5**	4	8
3	4	**7**	12	16
0	4	**4**	0	4

Table 4.1: Some solutions to the Node Size Problem.

4.2.5 A Concrete Graph Specification.

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

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

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

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

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

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

4.2.6 A Short Example

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

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

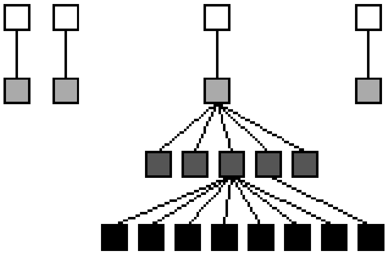


Figure 4.2: A simple inheritance tree.

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

Once we have displayed the tree as we see in Figure 4.2 we apply size and color metrics to the nodes. The use of position metrics is not possible here, as the position of the nodes is intrinsically defined by their logical position in the tree. As the nodes represent classes, we use class size metrics. The width and height of the nodes render the number of methods (NOM) while the color renders the number of attributes (instance variables).

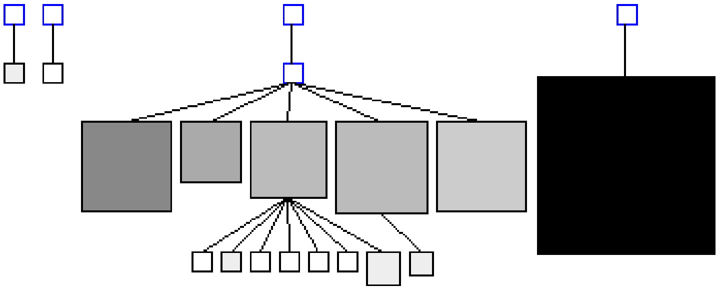


Figure 4.3: An inheritance tree that makes use of size and color metrics.

Once the tree is rendered as in Figure 4.3 we can start interacting with the graph. We can freely move nodes around, delete them, inspect them (i.e. browse the underlying classes), filter out parts, etc.

In fact, if we left out the interactive part, the amount of useful information that we could extract would be limited to the display in Figure 4.3.

4.3 Our Approach

We listed above some properties that graphs for reverse engineering should possess. In this section we see how our approach behaves in their respect.

- *Simplicity and Quality*. We chose to visualise the basic object-oriented entities (classes, methods, attributes) as colored rectangles. The advantages of this choice is discussed in Section 4.2.3, where we include the representation of metrics in our visualisations. The basic relationships that occur between those entities (inheritance, invocation, access) are rendered as edges between the entity rectangles.

- *Quantity*. We enabled our tool CodeCrawler to be able to display information at will, i.e. edges can be removed, nodes can be removed and we can zoom into parts of the graph, while removing the other parts from our selection. The quantity of displayed code is thus arbitrary and gives us an important freedom of choice.

- *Colors*. What we think is a good approach, is to use *color shades*. This has the advantage that numerical information can be transmitted by colors. While numerical values are easy to transmit by size (bigger numerical values are mapped on bigger things), we can thus also map a numerical value into a certain color shade. For the work done during this project we chose to use color shades between white and black, going over all shades of gray. The higher the numerical value the *darker* the mapped color is. Thus light gray means ’smaller’ than dark gray. Although this is a good way to represent a supplemental numerical value, there are a few considerations to be made:

  - Since the perception of a color tone is less precise than the perception of size, the color cannot reflect small differences in the numerical values, this technique is thus only useful for the detection of considerable color shade differences.
  - The linear mapping from an interval of values into a predefined interval (like 0..1), which itself defines the color shade, is based on the range of the first interval. If the first interval is stretched by extremely big or small values, the whole mapping function is distorted. These big values are normally rare, but they can make all the other ’average’ values to be mapped into a small color shade interval and will thus look similar. We found a partial workaround for this problem, by enabling the use of a logarithmic and an exponential mapping function. Although this is of some help, both of them have a major drawback, which is that the mapped values are distorted because of the non-linearity of the mapping functions.

- *Scalability*. We found out graphs which are scalable and which work with high numbers of nodes. We saw in a real industrial case study of considerable size that our approach worked very well. We describe that experience in Chapter 8.

- *Interactivity*. We provide interactivity in many aspects: the nodes in the graph can be moved around freely and removed from it. The underlying software entity can be inspected, browsed, and queried. This adds to our solution a playful nature which we think is primarily responsible for the intuitive approach.

- *Metrics*. In Section 4.2.3 we have discussed the way metrics are used with our approach. The way we use metrics is direct and intuitive and easy to grasp.

4.3.1 Conclusion

We have seen that our approach fulfils most of the above mentioned requirements. However, we emphasise that simplicity comes at a certain cost: Using an elaborate algorithm could be of more use, the same goes for the metrics.

We now have seen that there is a substantial difference in the above mentioned four levels of conception of a graph.

If we consider that we have a certain number of graphs and layouts at out disposition, and that we can apply a set of metrics on them, we can easily estimate that the number of possible graphs is huge. The heart of this whole work is the discovery of so-called *useful graphs*. We refer to such graphs as useful is we can use them for reverse engineering a system. However, we are not only trying to build a collection of useful graphs which can produce useful insights on systems. We are forcing the whole concept of reverse engineering as a creative, investigative process. The interactive part is tightly connected to the reverse engineering process and is sometimes crucial to the usefulness of a graph.

Chapter 5

CodeCrawler

5.1 Introduction

CodeCrawler is the program implemented during this work and which generated all graphs in this document. The program has been written entirely in Smalltalk in the Visual Works 3.0 environment, which comes from ObjectShare, and which is free for non-commercial use[1]. The framework that CodeCrawler uses for the graphical output is called HotDraw[BRAN 95] and is treated in Section 5.4. The framework used to generate metamodel information is the Smalltalk implementation of the FAMIX metamodel called Moose and is described in Section 5.3. CodeCrawler does not work without the above mentioned frameworks. We include a section on the implementational details of CodeCrawler at the end of this chapter.

5.2 Requirements and functionality

When the implementation of CodeCrawler started, we had some functionalities and properties in mind which we thought would be necessary for the program to have:

- **Language Independency**. To preserve maximum flexibility, we thought it necessary for CodeCrawler to be based on a language independent metamodel. One of the major drawbacks of many software visualisation tools, is that they run only under certain environments and they can display only certain languages. However, reverse engineering is a field independent of the used language (there are systems which have been implemented in more than one language!). CodeCrawler has been successfully tested on systems written in Smalltalk, C++ and Java. Ada is soon to be expected.

- **Platform Independency**. The VisualWorks 3.0 non-commercial environment, in which CodeCrawler has been written is platform independent, as it is based on a virtual machine. It is available for the following platforms: Unix, Windows 95/98/2000/NT, Mac OS and Linux. This increases the utility of such a tool, as it doesn't have to be ported explicitly to other platforms.

- **Interactivity**. Through means of direct-manipulation interfaces, we want to give the user of CodeCrawler the chance to *navigate* through the source code. This

[1] Please consult http://www.objectshare.com on the Internet for additional information.

interaction with the graphs can also deepen the understanding of the displayed information.

- **User friendliness**. For the boundaries of such a work this prerequisite does not seem important, since the main part should be the graphs and they can be very well generated without dialog boxes, etc. However, with the program getting more and more complex, it soon became necessary for CodeCrawler to provide the means to rapidly change the parameters.

- **Flexibility**. We saw the need for an implementation which is easy to extend, and where graphs, layouts and metrics could be added without much fuss. This can also seen as a countermeasure against the short life cycle which such tools tend to suffer and which we wanted to prevent.

- **Availability**. CodeCrawler is free to be used and available on the Internet at the following URL:
`www.iam.unibe.ch/∼scg/Archive/Software/CodeCrawler/`

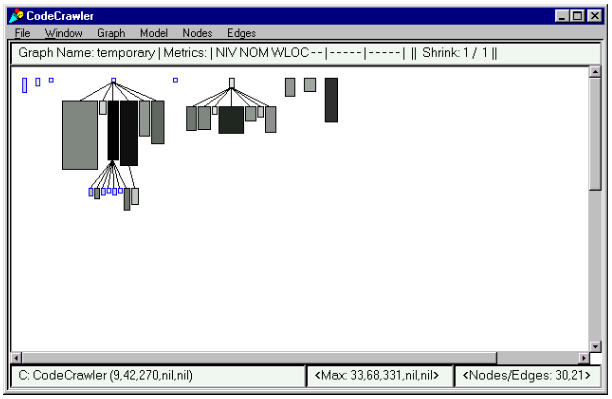
Nearly all of the above goals have been reached to our satisfaction. We will now have a look at the program and its user interface. CodeCrawler has one main window, and several secondary windows, which are

1. **The main window**
2. **The graph generator** consisting of
   (a) The graph panel
   (b) The metrics panel
   (c) The options panel
   (d) The repository panel
3. **The model builder**
4. **The selection viewer**
5. **The control panel**

We will now have a closer look at those components:

1. **The Graph Window.** This window is where the graphs are displayed after the user has chosen what kind of graph he wants to display. In Figure 5.1 we can see the main window with an inheritance tree display of CodeCrawler . The graphical display is interactive and the user can click on the nodes and edges and extract further information. He can freely delete nodes and edges, highlight them, etc.

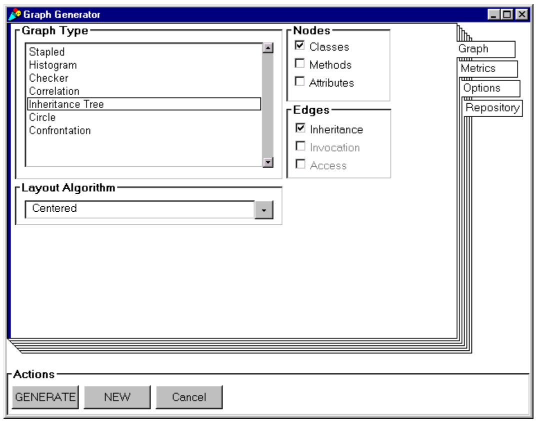
2. **The Graph Generator.** This window is a sort of control center where the user can select and set the parameters for the graph to be generated. Once the selections have been taken and accepted, the graph will be displayed on the main window. It should be noticed that while we implemented error-catching routines, the user is not protected from generating senseless graphs. The generator consists of four panels which are listed and described below.



Graph Name: temporary | Metrics: | NIV NOM WLOC - - | - - - - - | - - - - - | || Shrink: 1 / 1 ||

C: CodeCrawler (9,42,270,nil,nil) | <Max: 33,68,331,nil,nil> | <Nodes/Edges: 30,21>

Figure 5.1: CodeCrawler's main window.



Graph Type
Stapled
Histogram
Checker
Correlation
Inheritance Tree
Circle
Confrontation

Nodes
☑ Classes
☐ Methods
☐ Attributes

Edges
☑ Inheritance
☐ Invocation
☐ Access

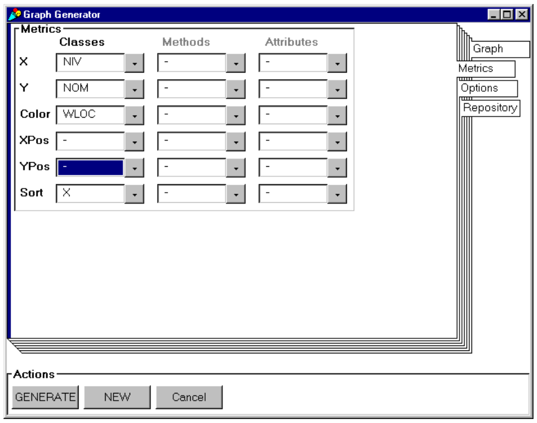
Layout Algorithm
Centered

Figure 5.2: The generator's graph selection panel.

In the graph selection panel (Figure 5.2) the graph and layout can be selected, and we can also choose which kind of entities and relationships we want to display. In the metric selection panel (Figure 5.3) we can assign metrics to the entities we want to display. CodeCrawler supports all metrics listed in Chapter 3. Further options can be chosen in the options panel (Figure 5.4). The graph repository panel we see in Figure 5.5 serves as a database for generated graphs, which can be loaded and saved from/to disk. If we select a graph in the repository all options that can be selected in the other panels will be automatically set.

3. **The Metamodel Builder**. In this window the user can build a metamodel, either by selecting Smalltalk classes and building it on the spot, or by loading a metamodel from a CDIF file. A model which has been built can also be saved to disk as a CDIF file. As a further and very important note: the meta model is **not** part of a graph. The graph is always built upon parts of the metamodel, but saving a graph does not also save the meta model.

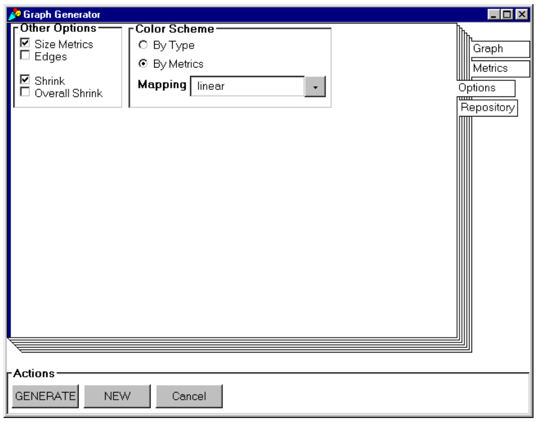
4. **The Selection Viewer**. Since we don't want our graphs to be built on the whole model, but often only on parts of it, this window provides the functionality to filter out parts of the model or just a way to list all entities present in the model.



Metrics

	Classes	Methods	Attributes
X	NIV	-	-
Y	NOM	-	-
Color	WLOC	-	-
XPos	-	-	-
YPos	-	-	-
Sort	X	-	-

Figure 5.3: The generator's metric selection panel.



Other Options

☑ Size Metrics
□ Edges
☑ Shrink
□ Overall Shrink

Color Scheme

○ By Type
● By Metrics

Mapping linear

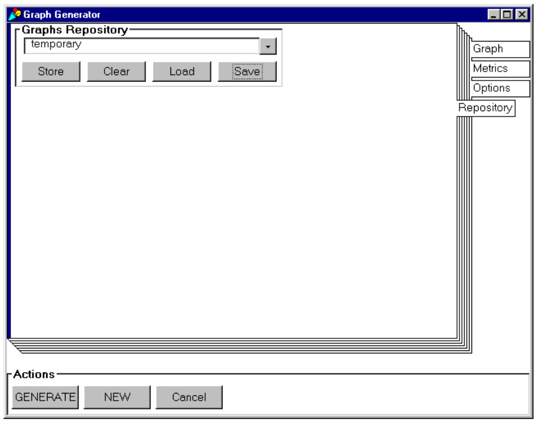
Figure 5.4: The generator's options selection panel.

Removing an entity from this selection, for example a class, causes CodeCrawler to remove all entities which are associated with it (i.e. the methods, the attributes, etc.). The selection can be extended and shrunk at will. Removing nodes from a graph display on the main window can also involve removing the associated entity from the current selection, if the user wishes to do so.

5. **The control panel**. The control panel is an add-on to the main window. Here a few parameters regarding the graphical output can be inspected and changed.

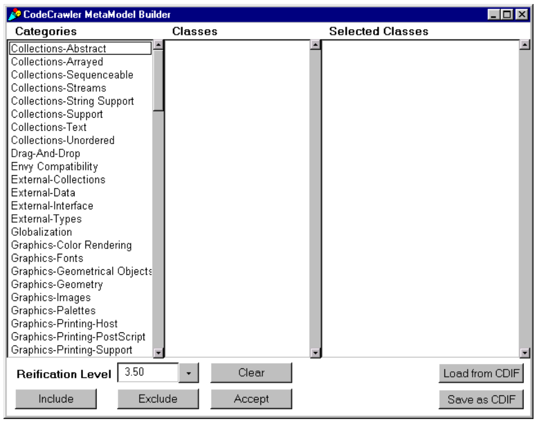
5.3 FAMIX

The implementation of the FAMIX metamodel written in Smalltalk is called Moose and has been developed at the University of Bern, Switzerland, by Dr. S. Demeyer and Dr. S. Ducasse. It is part of a European project called FAMOOS (Framework based Approach for Mastering Object Oriented Systems). The FAMOOS ESPRIT project 21975 investigates tools and techniques for transforming object-oriented legacy systems into frameworks. See *http://www.iam.unibe.ch/∼famoos/* for more information.



Graphs Repository: temporary

Figure 5.5: The generator's graph repository panel.



Reification Level 3.50

Figure 5.6: CodeCrawler's metamodel builder window.

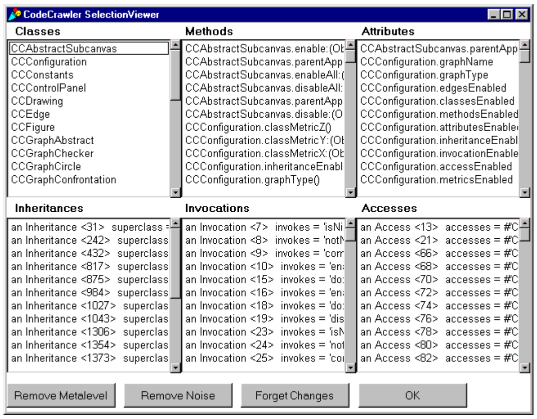
The model itself is a language-independent database of Object Oriented entities. Once a model has been built, we can make queries to the model and its entities. Suppose we have an entity representing a class. We can now ask this class to give us all its methods, attributes etc. The Moose framework has been written in Smalltalk. We can build models out of systems written in other Object Oriented programming languages than Smalltalk through an interface called CDIF.

In Chapter 3 we list the metrics that the Moose model is currently supporting.

The Data Model. CodeCrawler is based on an language independent representation of object-oriented source code, named FAMIX (FAMoos Information EXchange model, see [TICH 98]). FAMIX is defined in the context of the FAMOOS project and exploits meta-modelling techniques to make the data model extensible.

The data model model comprises the main object-oriented concepts –namely Class, Method, Attribute and Inheritance Definition– plus the necessary associations between them –namely Invocation and Access (see Figure 5.9).

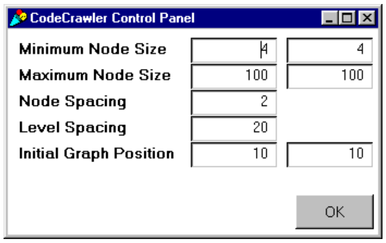
- **Advantage:** Due to the language independent nature of FAMIX, CodeCrawler



Classes	Methods	Attributes
CCAbstractSubcanvas	CCAbstractSubcanvas.enable:(Ob	CCAbstractSubcanvas.parentApp
CCConfiguration	CCAbstractSubcanvas.parentApp	CCConfiguration.graphName
CCConstants	CCAbstractSubcanvas.enableAll:(	CCConfiguration.graphType
CCControlPanel	CCAbstractSubcanvas.disableAll:	CCConfiguration.edgesEnabled
CCDrawing	CCAbstractSubcanvas.parentApp	CCConfiguration.classesEnabled
CCEdge	CCAbstractSubcanvas.disable:(O	CCConfiguration.methodsEnabled
CCFigure	CCConfiguration.classMetricZ()	CCConfiguration.attributesEnable
CCGraphAbstract	CCConfiguration.classMetricY:(Ob	CCConfiguration.inheritanceEnabl
CCGraphChecker	CCConfiguration.classMetricX:(Ob	CCConfiguration.invocationEnable
CCGraphCircle	CCConfiguration.inheritanceEnabl	CCConfiguration.accessEnabled
CCGraphConfrontation	CCConfiguration.graphType()	CCConfiguration.metricsEnabled

Inheritances	Invocations	Accesses
an Inheritance <31> superclass =	an Invocation <7> invokes = 'isNi	an Access <13> accesses = #'C
an Inheritance <242> superclass	an Invocation <8> invokes = 'notN	an Access <21> accesses = #'C
an Inheritance <432> superclass	an Invocation <9> invokes = 'com	an Access <66> accesses = #'C
an Inheritance <817> superclass	an Invocation <10> invokes = 'en:	an Access <68> accesses = #'C
an Inheritance <875> superclass	an Invocation <15> invokes = 'do:	an Access <70> accesses = #'C
an Inheritance <984> superclass	an Invocation <16> invokes = 'en:	an Access <72> accesses = #'C
an Inheritance <1027> superclas	an Invocation <18> invokes = 'do:	an Access <74> accesses = #'C
an Inheritance <1043> superclas	an Invocation <19> invokes = 'dis	an Access <76> accesses = #'C
an Inheritance <1306> superclas	an Invocation <23> invokes = 'isN	an Access <78> accesses = #'C
an Inheritance <1354> superclas	an Invocation <24> invokes = 'not	an Access <80> accesses = #'C
an Inheritance <1373> superclas	an Invocation <25> invokes = 'co	an Access <82> accesses = #'C

Figure 5.7: CodeCrawler's selection viewer window.



Minimum Node Size	4	4
Maximum Node Size	100	100
Node Spacing	2	
Level Spacing	20	
Initial Graph Position	10	10

Figure 5.8: CodeCrawler's control panel.

can in principle be applied on all object-oriented implementation languages. Being language independent is an important criterium, as it is an argument supporting the "Help Yourself" approach, because chances are higher that the effort in tool construction will pay off.

- **Limitation:** In practice, we must limit ourselves to languages that can be parsed into a FAMIX representation. At the time of writing, these are SMALLTALK, Java and a subset of C++. Ada is soon to be expected.

Metrics. The measurements of given source code entities are attached to the FAMIX counterparts (see Figure 5.9). Thus, a class entity knows about its number of methods and number of attributes; a method entity knows about its number of statements, etcetera. Most of the metrics listed in Chapter 3 can be derived from the data model itself, thus are language independent. However, a few of them (i.e.; number of statements in method body - M-NOS; number of methods overridden & extended - NMO & NME) require a language independent interpretation.

- **Advantage:** Since most of the metrics applied in CodeCrawler are language independent in nature, a lot of the CodeCrawler can be reused across different implementation languages, again supporting the principle of ”Help Yourself”.

- **Limitation:** A considerable part of the reverse engineering capabilities – especially analysing the quality of the inheritance tree – is based on the language dependent metrics. Thus, if one wants to reuse a hybrid metrics-visualisation

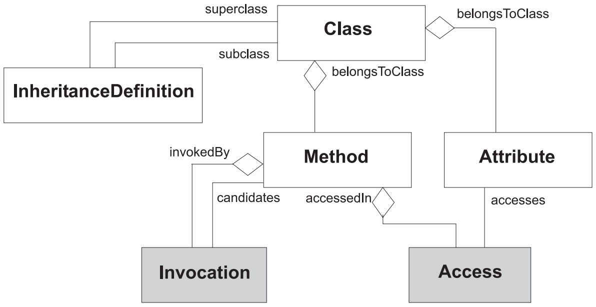


Figure 5.9: The FAMIX Data Model underlying CodeCrawler

toll across implementation languages, some language dependent customisation will be required.

5.4 HotDraw

HotDraw is a two-dimensional graphics framework for structured drawing editors that is written in Smalltalk by John Michael Brant [BRAN 95].

A HotDraw application edits drawings that are made up of figures. Figures are graphics elements such as lines, boxes, and text, and they can represent other objects. A drawing editor built from HotDraw (such as CodeCrawler) contains a set of tools that are used to manipulate the drawing. When a figure is selected by the selection tool, it presents a set of handles. Manipulating a handle changes some property of its figure or performs some action. For further information on the HotDraw framework, which is still being maintained, see also [BECK 94, JOHN 92, BRAN 95].

5.5 Implementation

This chapter deals with a few aspects regarding the implementation of CodeCrawler.

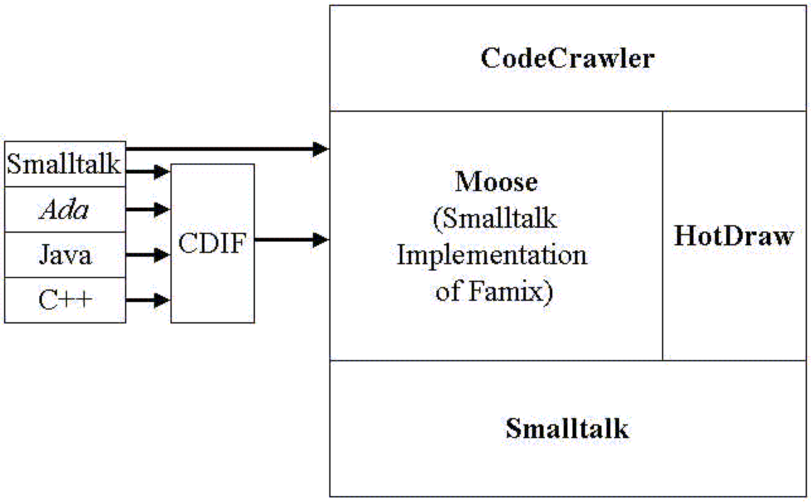


Figure 5.10: The general structure of CodeCrawler's logic.

In Figure 5.10 we can see the way CodeCrawler has been built. The language independent metamodel developed during the FAMOOS project is called FAMIX, and based on the specifications a Smalltalk implementation called Moose has been written. The entities can be stored into Moose either by filing in a CDIF file or by directly generating the model out of Smalltalk classes. CodeCrawler itself is also implemented in Smalltalk. The graphical framework used for the output on the screen is called HotDraw.

The general idea of CodeCrawler is to have a drawing on which nodes are represented. The nodes are eventually connected to each other by edges. Each node represents a language independent metamodel entity, which can be a class, a method or an attribute. Each edge represents a relationship, which can be inheritance, invocation or access.

5.5.1 Attachment To The FAMIX Model

The underlying metamodel serves primarily as a database for CodeCrawler. It returns the entities CodeCrawler wants to represent and is able to calculate the metrics and respond to queries. The attachment is present in three places:

- CCSelection. This class represents a subset of the model. This enabled us to filter out unwanted entities without tampering with the actual model.

- CCService. This class needs to know what metrics can be calculated for the entities present in the model. Each time a new metric is added to the model it has to be registered in CCService. A major change in the future will be that this class fetches the appropriate metrics (i.e. language specific) on its own from the model.

- CCNode. This class holds a direct reference to the underlying metamodel entity it represents and can query the model in case this is needed to return some information.

5.5.2 Attachment To HotDraw

HotDraw is the graphical framework upon which CodeCrawler is based. HotDraw itself is not changed in any way, because CodeCrawler subclasses everything it needs. At this time CodeCrawler subclasses the following HotDraw classes:

- DrawingEditor. This is done through the class CodeCrawler, which is the main application class.

- Drawing. The corresponding CodeCrawler class is named CCDrawing and returns some supplemental information on displayed nodes and edges.

- RectangleFigure. The subclass is named CCFigure in CodeCrawler. It is described in the next section.

- Tool. The class CCTool implements the method which is responsible for displaying the node specific information in the lower left corner of the main application window, when the mouse pointer is floating above a CCFigure in the drawing.

5.5.3 Important Classes

CodeCrawler

CodeCrawler is the main application class. It is a subclass of a HotDraw DrawingEditor. However, it implements much more functionality, like all the functionality accessible through the menus. It holds references to all subwindows like the generator and the control panel. It also holds a direct reference to the current graph.

CCGraphAbstract

This class defines all functionality common to the graphs implemented in CodeCrawler. It holds collections of nodes and edges. A graph is built upon the current state of the class CCSelection. This class needs to be subclassed if a new graph is to be added to CodeCrawler. CCGraphAbstract can't be instantiated, because it returns no layout algorithms. A concrete graph class can have several layout algorithms, which have to be registered in the class CCService.

CCNode and CCTreeNode

A CCNode represents a node in a CodeCrawler graph. A CCTreeNode is a subclass which implements some added functionality which is needed in a tree graph. In Figure 5.11 we can see that a node directly references its *figure* and the metamodel entity it

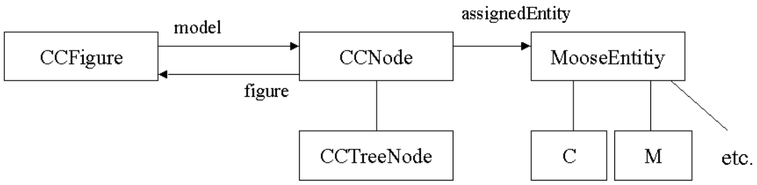


Figure 5.11: The class CCNode and its partners.

represents (*assignedEntity*). The assigned entity can be any Moose entity. A CCNode is part of a *graph* and can return functionality like menus, operations like highlighting etc.

CCFigure

This class is a subclass of a HotDraw RectangleFigure. It is the graphical representation of a node class and directly references it through the variable *model*, as we can see in Figure 5.11. A few operations which can be done on a CCFigure are reflected directly on its assigned CCNode or CCTreeNode object.

CCSelection

This class is responsible for managing six collections of objects, which are the classes, methods, attributes, inheritances, invocations and accesses present in the current model. Since the user will not want to interact on the whole current model all the time, this class presents a subset of the model which can be changed without tampering with the actual model. This selection can be filtered either programmatically (by applying filters) or interactively (by removing nodes from the drawing whose entities are then removed from the selection). It is always possible to trace back the selection to its original state, which is identical to all entities in the model. At this time this class is a singleton, but this is certain to change in the future to allow the use of multiple selections.

CCConfiguration

This class holds all attributes which are necessary to identify a graph configuration. Each time the user interacts with the graph generator by selecting the properties of the graph he'd like to have, when at the end the user decides to build a new configuration, all possible options from the generator are put into a new configuration object. Such a configuration object is also what gets saved when we save a graph in the repository subcanvas. Each time a CodeCrawler graph configurations file (*.tln) is loaded into CodeCrawler the repository is filled with configuration objects. When such a configuration is selected in the repository, all possible options in the generator are set accordingly.

CCService

This class provides services to many classes in CodeCrawler. It is a singleton class, and should be cleared each time its implementation changes. It can return the graphs and layouts implemented, the metrics and where they can be applied, etc.

CCConstants

The constants class has a somewhat deceiving name. In fact it holds all values which are used in CodeCrawler for layout. These are are divided into three dictionaries named *colors, magnitudes* and *points*. The constants class is a singleton and should be cleared as soon as its implementation changes. The magnitudes and the points are used in the layout algorithms and can be changed interactively through the control panel.

Chapter 6

Useful Graphs

6.1 Introduction

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

Before we start with our classification, we want to make a few statements about the graphs discussed below:

- Not every graph here is effective on our case studies. This is due to the fact that since some of these graphs serve to detect problems, sometimes they can’t do that because the problem which a graph should detect is not present in the system.
- We divided the graphs treated here into 4 distinct groups, which we call *class*, *method*, *attribute* and *class internal*. The names indicate which kind of entities are displayed in the graphs. Class internal treats the special case where methods and attributes are displayed at the same time.
- The graphs presented here are only a selection of those discovered by us which we judge to be useful. However, we preferred to make a selection to keep the amount of presented graphs reasonable.
- A general rule for all graphs presented here is that each one has drawbacks which another one may be able to alleviate or remove at the cost of developing itself in turn one or more disadvantages.

Structure. The structure of this chapter is the following:

- **Useful Graph Structure.** In Section 6.2 we explain the structure which we adopted to discuss each useful graph.
- **Cases Studies.** In Section 6.3 we have a short presentation of the two case studies which we used to test CodeCrawler on.

- **Layout Algorithms.** In Section 6.4 we present each layout algorithm which is mentioned in this chapter and explain their properties.

- **Useful Graphs.** The rest of the chapter discusses each useful graph in detail and presents the results when applied on our case studies.

6.2 Graph Structure

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

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

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

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

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

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

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

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

6.3 Case Studies

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

- Availability. Both case studies are public domain and can be downloaded freely. With this point we can ensure that the results are reproducible.

- Size. We chose two case studies which can be termed as being of an *average size* and are representative of medium-sized standalone applications. We think that very small applications can’t reflect results properly because the purpose of most graphs is coping with complexity, which in such cases is not necessary. On the other hand, if we had chosen very big applications, it would have been hard to present results in a concise manner, because many graphs can be applied in

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

Table 6.1: An overview of the size of our case studies.

various areas and at various levels of granularity. However, we present some experiences we had with very large systems in Chapter 8.

- Level of maturity. We chose one very mature application which has gone through some refactorings and redesigns, and another one which has been developed in a rush and which has yet to undergo its first redesign. We did this to see if the results of our experiments would differ and in what way they would do that.

6.3.1 Refactoring Browser

The Refactoring Browser is a widely used tool for the implementation of Smalltalk programs [ROBE 97]. We took it as a case study because it is an application which has gone through several refactorings and redesigns and has been written by some very experienced programmers. This quality of implementation should thus be visible in such a system. It is a medium sized application as we can see in table 6.1.

6.3.2 Duploc

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

6.4 Layout Algorithms

6.4.1 Introduction

This section is dedicated to the graphs and layouts we have selected to implement in CodeCrawler. We discuss the properties, advantages and drawbacks of each one of them. We include this here because they are mentioned throughout the remainder of this chapter. As some layouts are not being used in this chapter we decided to put them in the appendix in Chapter A.

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

In Table 6.2 we have an overview of all graphs and their properties supported by CodeCrawler. The circle graph is discussed in Chapter A.

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

Table 6.2: CodeCrawler's graph layouts.

The 'Metrics' column specifies how many metrics can be rendered by the graph. 5 means that the a single node can render 5 metrics at the same time. $3 + 3$ means that two separate groups of entities and metrics can be defined. The 'Entities' column refers to the kind of entities the graph can be applied upon: C for class, M for method and A for attribute[1]. The 'Scope' column specifies if the graph can be applied to the complete (sub)system or only to some entities like a class or a method. The 'Sort' column indicates if a sorting of the nodes according to a certain metric measurement can enhance the usefulness of the graph in question.

[1] The limitation to these three types of entity is due to the current implementation of the Moose model. Future implementations of it may include supplemental entities as we point out in Chapter 9.

6.4.2 The Tree Graph

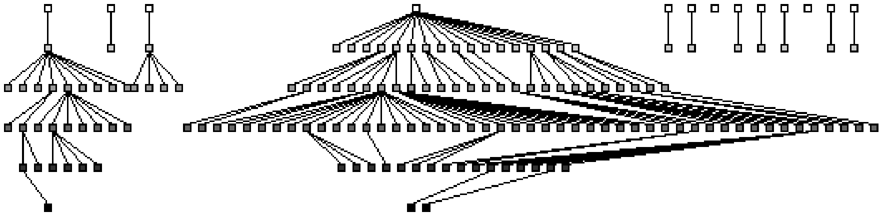


Figure 6.1: A tree graph of a system.

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

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

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

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

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

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

6.4.3 The Correlation Graph

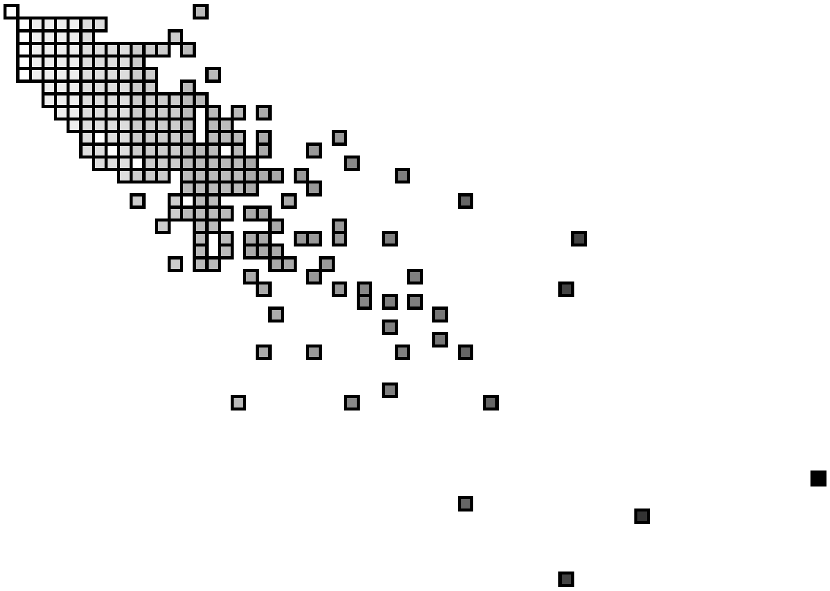


Figure 6.2: A correlation graph of method nodes using LOC and NOS as position metrics.

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

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

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

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

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

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

6.4.4 The Histogram

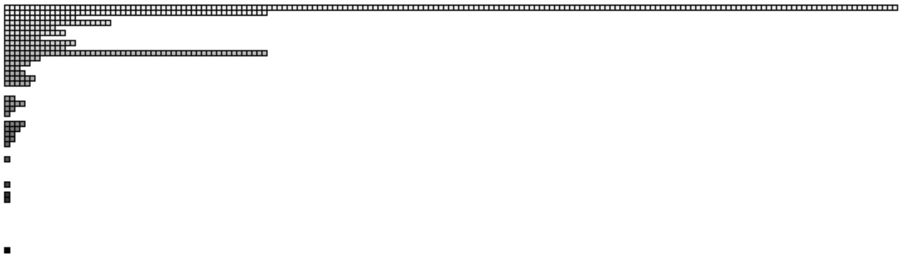


Figure 6.3: A horizontal histogram.

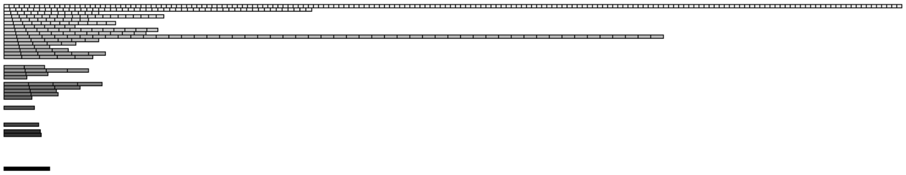


Figure 6.4: A horizontal histogram using the size addition layout

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

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

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

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

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

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

6.4.5 The Checker Graph

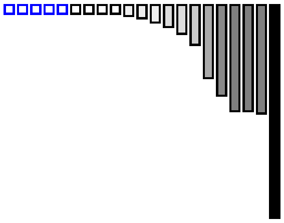


Figure 6.5: A checker graph using a sorted horizontal layout.

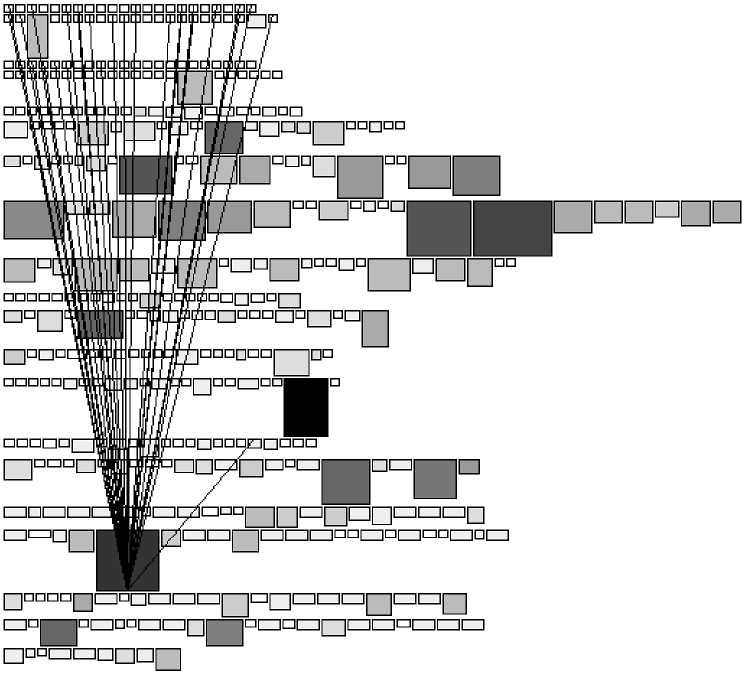


Figure 6.6: A checker graph using a quadratic layout with method nodes and invocation edges.

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

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

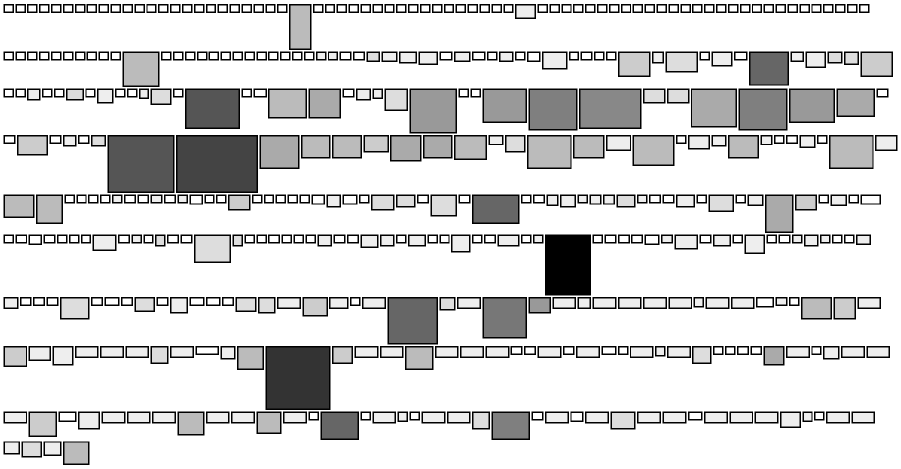


Figure 6.7: A checker graph using a maximal space usage layout.

Layouts. The first layout kind is called *horizontal* and *vertical*. We just place the nodes next to each other. We see such a layout in Figure 6.5 [2]. Because this wastes a lot of space, we introduced the *quadratic* layout which tries to lay out the nodes to make them form a rectangle, whose width is dependent of the number of displayed nodes. The graph which makes the best use of space is called *maximal space usage*, which tries to put as many nodes on the visible part of the drawing as possible.

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

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

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

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

6.4.6 The Stapled Graph

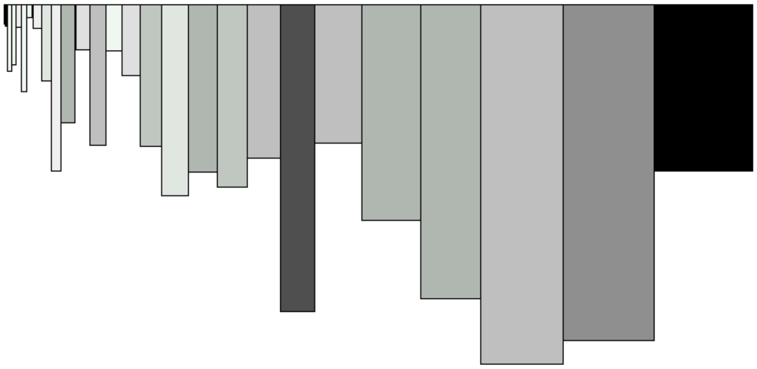


Figure 6.8: A stapled graph of class nodes.

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

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

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

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

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

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

6.4.7 The Confrontation Graph

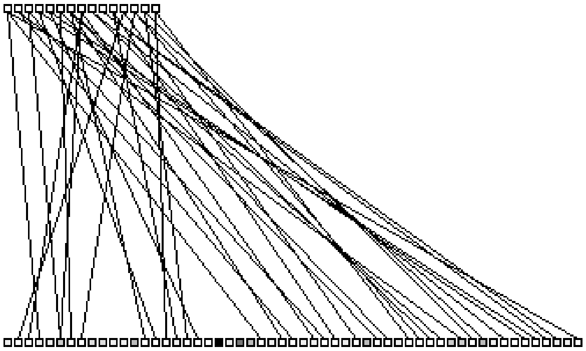


Figure 6.9: A confrontation graph using an horizontal layout

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

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

Layouts. There are two possible displays. The first, called either *horizontal* or *vertical* displays on one row (column) the attributes and on the other one the methods. We can see such a layout in Figure 6.9. However, since in a class often the number of methods is much greater than the number of attributes, and the graph very soon gets larger than the screen, we introduced the *three row* layout. In this case the attributes are in the middle row, while the methods are in the upper and lower row.

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

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

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

6.5 Class Graphs

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

1. Those which serve primarily for system understanding. They work at a higher abstraction level, and in some cases can only return a general statement about the system. Problem detection is secondary in such graphs and in some cases not even possible. The following graphs fall under this category:
   - SYSTEM COMPLEXITY, Section 6.5.1.
   - SYSTEM HOT SPOTS, Section 6.5.2.
   - WEIGHT DISTRIBUTION, Section 6.5.3.
   - ATTRIBUTE ORGANISATION, Section 6.5.4.
   - ROOT CLASS DETECTION, Section 6.5.5.
2. Those which primarily address problem detection, and secondarily program understanding. They must be applied on subsystems, rather than full systems. We list the following:
   - SERVICE CLASS DETECTION, Section 6.5.6.
   - COHESION OVERVIEW, Section 6.5.7.
   - METHOD PROTOCOL USAGE, Section 6.5.8.
   - SPINOFF HIERARCHY, Section 6.5.9.
   - INHERITANCE IMPACT, Section 6.5.10.
   - INTERMEDIATE ABSTRACT, Section 6.5.11.

6.5.1 System Complexity

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

General Idea: This is one of the first graphs that should be applied to a system. It is an overview of the inheritance hierarchies of a whole system. This graph can give clues on the complexity and structure of the system (how many classes are present?), as well as information on the use of inheritance in the system (how deep do the hierarchies go and is the system in general flat or deep?). If we furthermore apply some class complexity metrics we can extract some more information. In this case we use as size metrics NIV and NOM [3], while for the color we choose WLOC. The detection of aberrant classes is now made easy: we can see if there are *very large classes*, *small classes* or even *empty classes*.

Results with the Refactoring Browser: In Figure 6.10 we see the SYSTEM COMPLEXITY graph applied on the Refactoring Browser. It shows few stand-alone classes and a few deep hierarchies. The first thing that strikes the eye is the class *BrowserNavigator* (A) which has a huge number of methods (175) and lines of code (1495) compared to the other classes present in the system. At the same time it only has one instance variable (this is the reason for its very narrow look). It may be a case for refactoring. If we take a look at the inheritance tree on the right side we can spot the class *BRStatementNode* (B) which is completely empty. When I asked the developers of the Refactoring Browser about this case, they told me that they were aware of the problem and that this class had been created to duplicate a hierarchy of another program. The same case can be spotted on one of the stand-alone classes *RefactoringError* (D) which is also empty. The next point of interest is the class *BRScanner* (C) which has the most instance variables (14) while it implements comparatively few methods (52). Perhaps this massive stand-alone class could be split up into subclasses. Another thing we can see is, that in the inheritance hierarchy in the middle of the graph, the root class *Refactoring* (E) is implementing by far the most methods, while there are quite a few very small classes deeper down the inheritance chain.

Results with Duploc: When we apply the SYSTEM COMPLEXITY graph on Duploc, we can spot the following in Figure 6.11: The system shows some very flat inheritance hierarchies, with many stand-alone classes which can have considerable sizes. This could mean that the system has not yet been refactored. There are three deep hierarchies, although in all three we can see that the main work is being done by the roots, which indicates top-heavy hierarchies. We also see that the main class *called*

[3] For an explanation on the metric acronyms used in this chapter please consult the tables in Chapter 3.

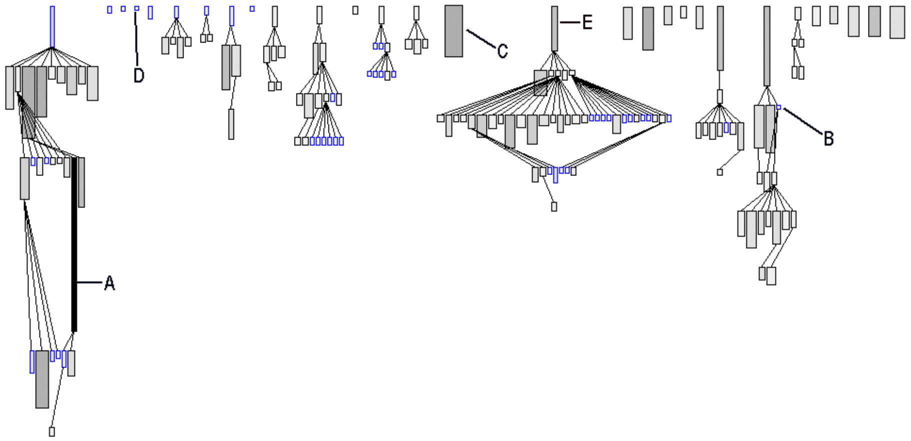


Figure 6.10: The system complexity graph applied on the Refactoring Browser using as size metrics NIV and NOM, and as color metric WLOC.

DuplocApplication (A) is very large and has only one very small subclass called it DuplocInformationMural[4]. Although *DuplocApplication* has the most methods and has the second most instance variables, the class with the most lines of code is *FastSparseCMatrix* (B). This class has only half the number of methods of *DuplocApplication* (74 vs. 130) but has nearly twice as much lines of code (1641 vs. 1060). Because of this we can already deduce that *FastSparseCMatrix* has some very long methods. The third point of interest are the classes on the left side (C): all of them are empty. These classes have become empty after being exported from the ENVY environment. The fourth eye-catch is the class *BinValueColoringModel* (D) on the right side. This class has the most instance variables (20), but only 52 methods. This may indicate that it is a service class which implements a lot of accessor methods. This supposition is being enforced by the light color value which is a sign for few lines of code (402), and is confirmed when we browse the source code of this class.

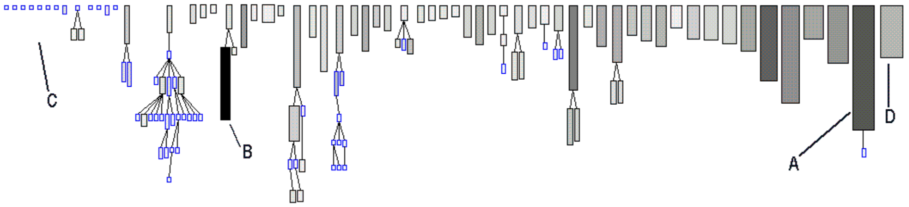


Figure 6.11: The system complexity graph applied on Duploc using as size metrics NIV and NOM, and as color metric WLOC.

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

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

Evaluation: This is certainly one of the first graphs that should be applied to a system, as it can return information on the structure and complexity of the subject system. However, it suffers one small drawback, which shows in very large systems: Sometimes the number of classes we want to display is so large that this graph takes several screens of place. It is difficult then to discern the outliers in the systems at one glance. The system hot spots graph discussed in Section 6.5.2 can counter this problem.

6.5.2 System Hot spots

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

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

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

Results with the Refactoring Browser: In Figure 6.12 we get a HOT SPOTS view on the Refactoring Browser. While in Figure 6.10 we had to search for the biggest and smallest nodes, this is made easy in this kind of graph because the nodes have been sorted: as before we can locate the class *BrowserNavigator* (A) and *BRScanner* (B). The sorting of the nodes makes it easy now to detect empty or very small classes, which find themselves at the top of the graph (D). Our attention is now also drawn to other classes like *BrowserApplicationModel* (C), which implements 38 methods while it defines no instance variable, which is visible by its flat shape. The view on the shape of the nodes is also facilitated, we can now detect classes like *MoveVariableDefinitionRefactoring* (E), which defines 6 instance variables while it implements only 7 methods (mainly accessors), giving it nearly a square shape.

Results with Duploc: The HOT POTS view on Duploc reveals also some information which could not be seen at first sight in Figure 6.11, as we see in Figure 6.13. We see Duploc has either very large classes (A)(B), or very small ones (D). We can also locate some classes with many instance variables (C). Two classes which could be interesting for further investigation because of are *DuplocCodeReader* (F) (32 methods, 17 instance variables) and *DuplocProgressMeter* (E) (15 methods, 9 instance variables): both classes have many instance variables and few methods, which could indicate service classes apt for refactoring.

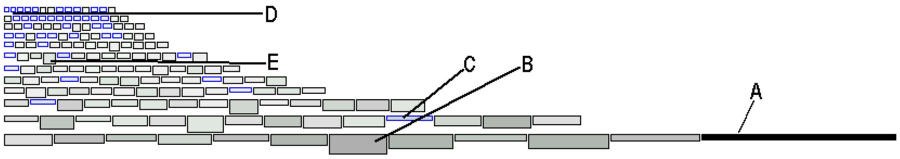


Figure 6.12: The system hot spots graph applied on the Refactoring Browser using as size metrics NOM and NIV, and as color metric WLOC. The nodes have been sorted according to NOM.

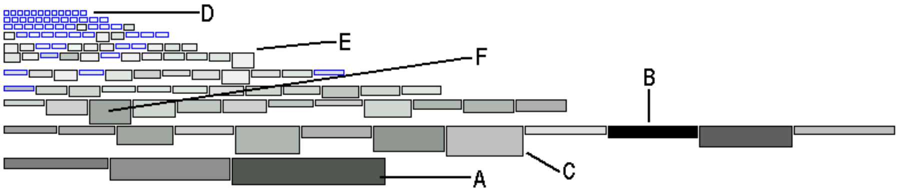


Figure 6.13: The system hot spots graph applied on Duploc using as size metrics NOM and NIV, and as color metric WLOC. Sort according to NOM.

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

Evaluation: The main drawback of the SYSTEM COMPLEXITY graph described in Section 6.5.1 is the fact that through the ordering of the nodes in tree structures we lose track of the size of the nodes all too easily. Only extreme cases strike our eyes. The SYSTEM HOT SPOTS graph described here makes this up through the sorting of the nodes and an ordering of them which reflects this sorting. However we lose the notion of inheritance in this case, since displaying the edges would mess up the view. A certain disadvantage of this graph is that the more nodes we display the more space is needed.

6.5.3 Weight Distribution

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

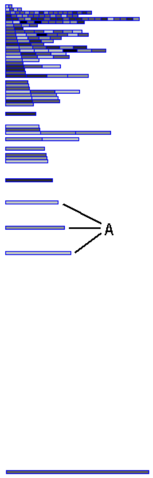


Figure 6.14: The weight distribution graph applied on the Refactoring Browser. As width and vertical position metric we use NOM, as color metric we use HNL.

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

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

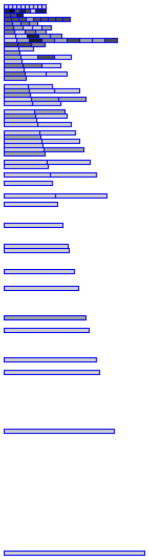


Figure 6.15: The weight distribution graph applied on Duploc. As width and vertical position metric we use NOM, as color metric we use HNL.

Results with the Refactoring Browser: The Refactoring Browser is an evenly distributed system, as we see in Figure 6.14: It's not possible to locate a majority of the dark or the light nodes on a certain area of the graph, although we can see there are three big classes marked as (A) high up the hierarchy.

Results with Duploc: Duploc is clearly a top-heavy system, as we see in Figure 6.15: The dark nodes are all very small (small NOM) and thus located on the top region of the graph. The big classes on the bottom of the graph are all very light (high up in the hierarchy). The system is thus to be classified as top-heavy, which is mainly due to its young age: Duploc has not yet undergone a reengineering or refactoring. It should be analysed on whether it's possible to introduce a supplemental abstraction level high up in the hierarchy.

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

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

6.5.4 Attribute Organisation

Graph	Checker, sort according to width metric.	
Scope	Full system, language specific: C++ and Java.	
Metrics		
Size	PriA (number of private attributes)	PubA (number of public attributes)
Color	ProA (number of protected attributes)	
Position	-	-

General Idea: One of the primary powers of object-oriented languages is certainly encapsulation; the ability to organize a class in such a way that it can be reused without having to know all its internal details. Because of this, in most languages the attributes can be defined as either private (only the class where they are defined can directly access them), protected (same as private, including all subclasses) or public (every class in the system can access it directly). The attribute organisation graph renders these three types of definition. It's a checker graph which uses as size metrics PriA and PubA and as color metric ProA. We chose ProA as color metric because several tests showed that most attributes are defined as either private or public, seldom protected.

We distinguish the following:

- Flat nodes are thus classes which are strictly private.
- Tall, narrow nodes are classes with many public attributes, which are very open to the system and which of course can be a danger.
- Dark nodes make use of protected definitions and it's often useful to check this detail with an inheritance tree display of the system: If the dark nodes happen to be stand-alone classes, there's a suspicious situation present: The use of protected attributes makes sense only in inheritance hierarchies, in stand-alone classes one could use private definitions just the same.

Possible Alternatives: The same principle works with the metrics PriM (number of private methods), PubM (number of public methods) and ProM (number of protected methods). In this case we base our observations on the methods implemented in the classes.

Evaluation: This graph should be used before we start examining subsystems. It can return a general statement about the system in general. It's also a good technique to apply it on each subsystem, to see if we can make out differences. As this graph is language specific and does not work for Smalltalk[5], we cannot show its application on our case studies and can thus not include a figure in this case.

[5] Smalltalk makes no distinction, it knows only protected attributes.

6.5.5 Root Class Detection

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

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

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

- If a node is positioned on the right side of the graph, while holding a vertical position near the top, this means that while the underlying class has a great number of descendants its direct subclasses are few. This is often the case when directly below a root class a supplemental abstraction level of classes has been introduced.

- If the node finds itself on the 45 degrees axis (it can't be further left because WNOC is always at least equal to NOC) and far away from the top of the graph, this means that the underlying class has a lot of direct subclasses. This is what we call a *flying saucer hierarchy* because the inheritance tree of this class resembles one.

- If a class node is positioned exactly along the 45 degrees axis this means that all its subclasses don't have subclasses themselves, and thus are leaf node classes in an inheritance tree.

Results with the Refactoring Browser: To make the effect of this graph more visible, in Figure 6.16 we see on the top left the root class detection graph while on the bottom right we see a display of two major inheritance trees. We see the class *Refactoring* (A) which has 43 descendants and 5 direct subclasses as root of major inheritance tree on the right side of the correlation graph. The other root class, *BrowserApplicationModel* (B) can also be identified on the right side of the graph. Two classes, *MethodRefactoring* (C) and *VariableRefactoring* (D), which are the heads of minor flying saucer hierarchies (14 and 13 direct subclasses) can be identified near the 45 degrees axis.

Results with Duploc: The results of this graph are somewhat deceiving in the case of Duploc, as its inheritance hierarchies are very flat. We can detect however two root classes, namely *PresentationModelControllerState* (A) and *PMCS* (B). In Figure 6.17

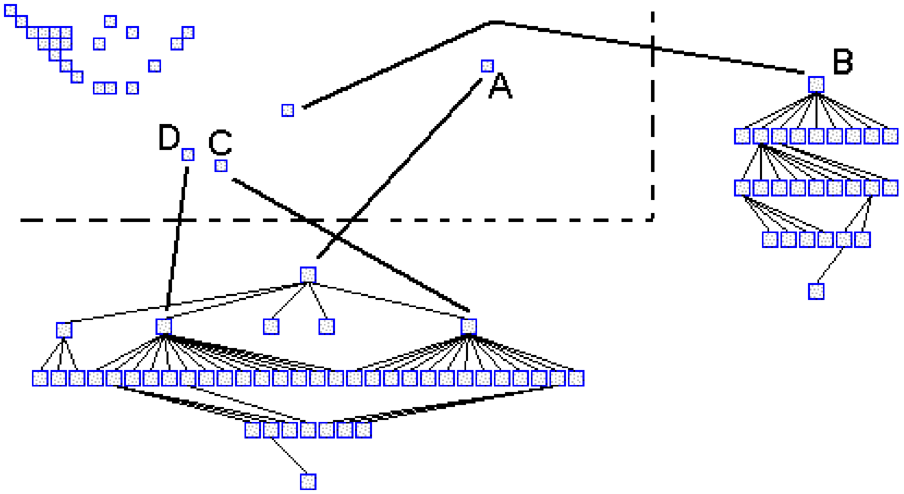


Figure 6.16: A root class detection graph applied on the Refactoring Browser. As position metrics we use WNOC and NOC.

we see where the detected root classes are located in one of the inheritance hierarchies of Duploc.

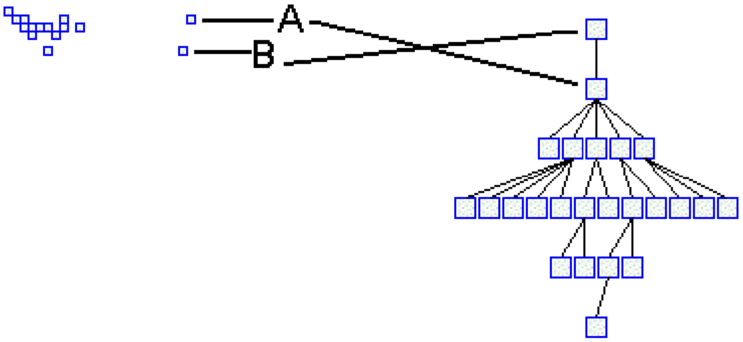


Figure 6.17: A root class detection graph applied on Duploc. As position metrics we use WNOC and NOC.

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

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

6.5.6 Service Class Detection

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

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

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

We can make out the following:

- If a class node breaks the staircase effect (by being too short) it is a candidate for a service class.

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

Results with the Refactoring Browser: In Figure 6.18 we selected a whole inheritance tree (26 classes) of the Refactoring Browser to be displayed in a SERVICE CLASS DETECTION graph. We see one huge class *BrowserNavigator* (A), which in fact is even bigger, but we cut it down because of space reasons. We see quite clearly that there is a certain tendency for a staircase which is severely broken in two places. The first service class candidate is *CodeTool* (B), which has 22 methods and 49 lines of code. A closer inspection reveals that the methods are mainly get/set-methods (accessors). The second candidate is *CodeModel* (C) with 40 methods and 136 lines of code. The name itself already reveals the service function this class is intended to have. As method splitting candidate we detect the class *ClassCommentTool* (D) which has only 7 methods but 89 lines of code.

Results with Duploc: We obtained the graph in Figure 6.19 by first applying the graph on the whole system and then by selecting a subset which looked interesting. We see there are some candidates for service classes: The class *CachedObservationData* (A) contains 20 methods for a total count of 50 lines of code. A closer inspection reveals it is truly a service class. Nearly the same ratio is visible in the classes *ComparisonMatrixBody* (B) (22/80), *PresentationModelControllerState* (C) (25/87) and *Ob-*

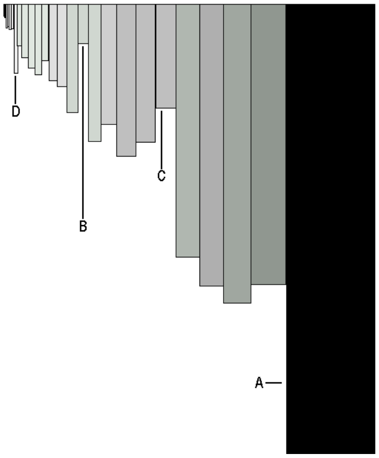


Figure 6.18: The service class detection graph applied on a subhierarchy of the Refactoring Browser. As width metric and sorting criterion we use NOM, the height metric is WLOC.

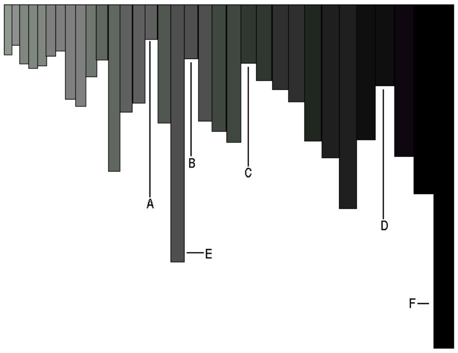


Figure 6.19: The service class detection graph applied on a subset of Duploc. As width metric and sorting criterion we use NOM, the height metric is WLOC.

servationOnRawSubMatrix (D) (30/122). Some classes tend to have overlong methods, namely *PMVSInformationMuralMode* (E) (22/396) and *DuplocCodeReader* (F) (32/530), and should be looked at for possible method splitting.

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

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

6.5.7 Cohesion Overview

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

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

- The flat nodes indicate that the methods of a class (the width indicates the number) do not access many times its instance variables. This is further emphasised by the small height (few instance variable accesses).
- The narrow and high nodes on the other hand, tend also to be very light colored. This case happens when the classes have many accesses but only few instance variables. This is mostly the case when the class defines an attribute which is then heavily accessed directly by its subclasses. This is not advisable because of the lacking encapsulation: a single access through an accessor which would then be invoked by other classes, instead of direct accesses on the attribute, would be much better.
- More or less rectangular nodes with darker color shades indicate a good cohesion inside those classes, although this is only provable after applying a class cohesion graph, which is described in Section 6.8.1.

Results with the Refactoring Browser: The resulting graph can be seen in Figure 6.20. The first thing we notice is that the nodes differ heavily in their shapes and colors. There are some white nodes that don't define instance variables (for example (A)) and because of this absence they can't have any instance variable access either. This is the reason for their flat shape. We also gather there are some empty or nearly empty ones (located around (F)). The class *BrowserNavigator* strikes once again the eye for its huge number of methods and its small number of instance variables (only one). The nodes (D) and (E) strike the eye for their narrow shape and light color: Both have few methods and instance variables, (1,2) and (2,1) respectively, while at the same time they have a huge number of accesses. The reason for this is that their variables are directly accessed by their subclasses. The class *BRScanner* (C) shows a great complexity and heavy access.

Results with Duploc: The graph in Figure 6.21 shows a few characteristics of Duploc: Many empty or nearly-empty classes (C), quite a few heavy-access classes (B) and (D) and a few very large classes, for example *DuplocApplication* (A). We see there are quite a few classes that could be interesting for inspection with a class cohesion graph and do that for one special case, the class *DuplocApplication* in Section 6.8.1.

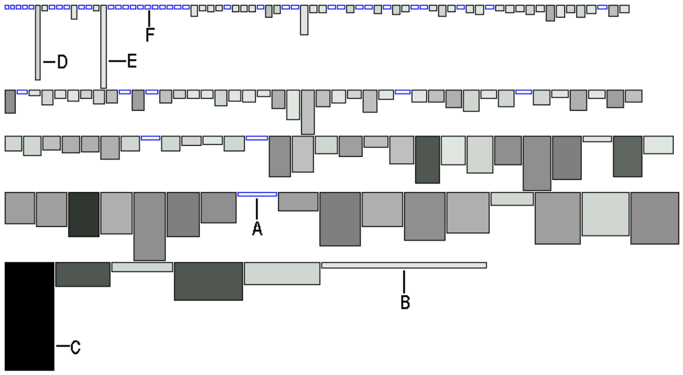


Figure 6.20: A cohesion overview graph applied on the Refactoring Browser. As size metrics we use NOM and WNAA. As color metric NIV is used.

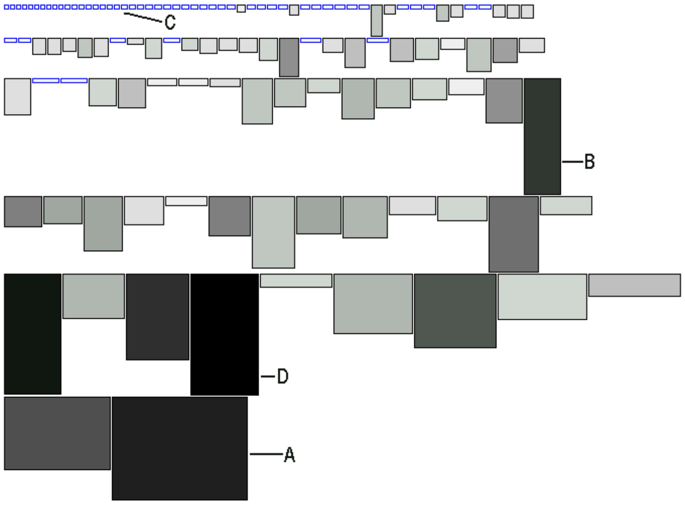


Figure 6.21: A cohesion overview graph applied on Duploc. As size metrics we use NOM and WNAA. As color metric NIV is used.

Possible Alternatives: None.

Evaluation: This graph can be seen as an *in-betweener*, because it comes after a graph for general overview and before a graph which treats class internals. The best result it can return is a collection of classes which we should further examine with a class cohesion graph, described in Section 6.8.1.

6.5.8 Method Protocol Usage

Graph	Checker, maximal space usage layout, sort according to width metric.	
Scope	Full system. Smalltalk specific.	
Metrics		
Size	NOMP (number of method protocols)	NOM (number of methods)
Color	*	
Position	-	-

General Idea: This graph can only be applied to programs written in Smalltalk, as it makes use of a language-specific metric, NOMP (number of method protocols). In Smalltalk the methods of a class are logically grouped in so-called protocols. This way it's possible to group all accessors, all initialize methods, etc. into a single protocol. This can make the understanding of a class easier, since we expect the methods located in the 'accessing' protocol to be accessors[6].

Two cases can harden the understanding of Smalltalk classes related to their logical structure: if a protocol contains too many methods (inexperienced Smalltalk programmers tend to do that) or if we have a great number of protocols and few methods (overzealous Smalltalk programmers tend to do that also). The third possible mistake is that a method is located in the wrong protocol or the protocol has a non-intuitive name.

The graph we present here can detect classes in which such a suspicious use of method protocols has been made. It shows the ratio between the number of methods (NOM) and the number of method protocols (NOMP) of a Smalltalk class. NOM is used for the height, while the width reflects the NOMP metric measurement. We are looking for nodes which have either a square shape (small ratio between NOM and NOMP) or very narrow, high nodes (big ratio).

Results with the Refactoring Browser: The first class to strike the eye in Figure 6.22 is once again *BrowserNavigator*, with 175 methods and 18 protocols. However, if we browse the class we see that the methods are distributed very irregularly on the protocols: the protocol 'private-class' contains itself 63 methods, which is very much. The same problems are encountered in the classes *BRScanner* (marked as (B) with 6 protocols, 49 methods, 25 of which are contained in the protocol 'private-scanning') and *InlineMethodRefactoring* (marked as (C), with 30 methods distributed on 3 protocols, but 27 of them in the same protocol). The inverse problem is not really present in the Refactoring Browser: although there are a few nodes which are square, they're small size shows they're not a problem. The only case we want to emphasise is the class *SystemNavigator* (D) which has 22 methods distributed on 12 protocols. Perhaps the number of protocols could be reduced to get a more severe grouping.

Results with Duploc: In Figure 6.23 we gather that there are few aberrant classes, as Duploc makes extensive and intelligent use of protocols. The only arguable point are the classes *FastSparseCMatrix* (A) and *DuplocApplication* (B), which have many

[6] The assignment of a method to a specific protocol is a decision taken by the programmer and it does not entail any constraints. The methods can be moved between the protocols freely.

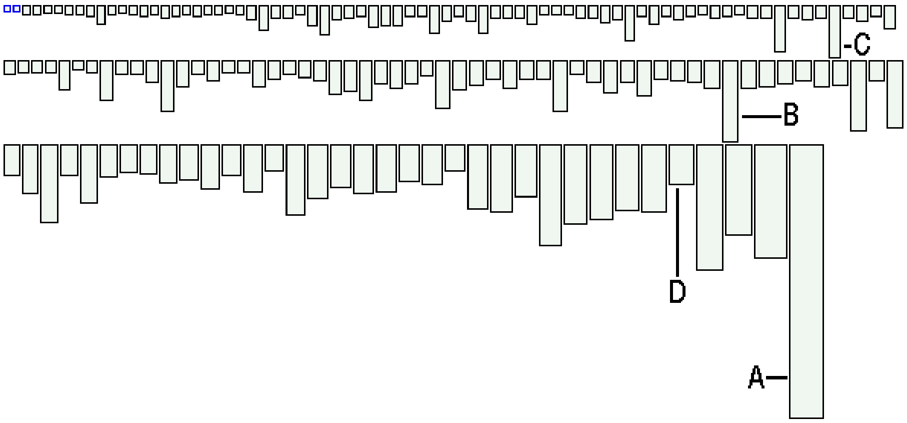


Figure 6.22: A method protocol usage graph applied on the Refactoring Browser.

protocols (21 and 19 respectively). The other hot spot is the class *BinValueColoringModel* (C), a very narrow yet tall class node. This brought up a new insight into the usefulness of this graph: It could also be used for the detection of service classes like in Section 6.5.6. As in Smalltalk it's common to put the accessors in the protocol 'accessing', this involves that a service class which has many accessors has relatively few protocols. It thus has a shape like in the case of *BinValueColoringModel*.

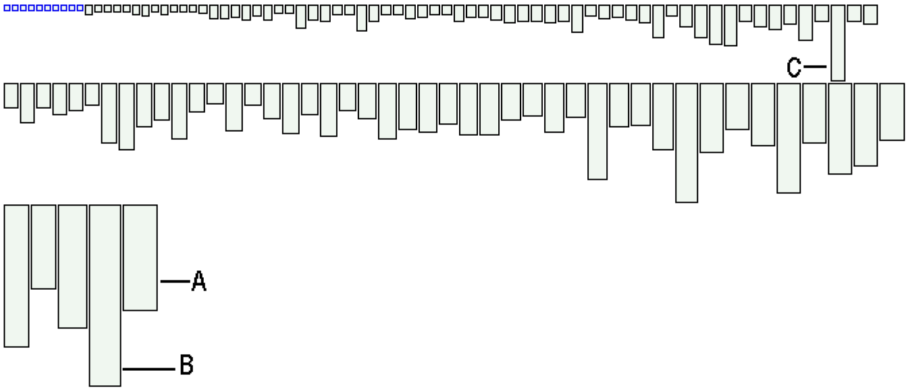


Figure 6.23: A method protocol usage graph applied on Duploc.

Possible Alternatives: We did not make use of the color metric. Its use could add supplemental information to this graph.

Evaluation: As this graph addresses a special problem, it should be applied to get supplemental information about subsystems.

6.5.9 Spinoff Hierarchy

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

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

We distinguish the following:

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

- Large, dark colored nodes. These are the *inheritance carriers*.

Results with the Refactoring Browser: In Figure 6.24 we see all inheritance hierarchies that make up the Refactoring Browser. We filtered out all stand-alone classes to get a clearer overview. We detect two cases of spinoff hierarchies:

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

2. The one with the class *Refactoring* (B) as root. Again two main inheritance threads are visible: The one consisting of *Refactoring* (B), *MethodRefactoring* (B11) and *ChangeMethodNameRefactoring* (B12). The other consists of *Refactoring* (B), *VariableRefactoring* (B21) and *RestoringVariableRefactoring* (B22).

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

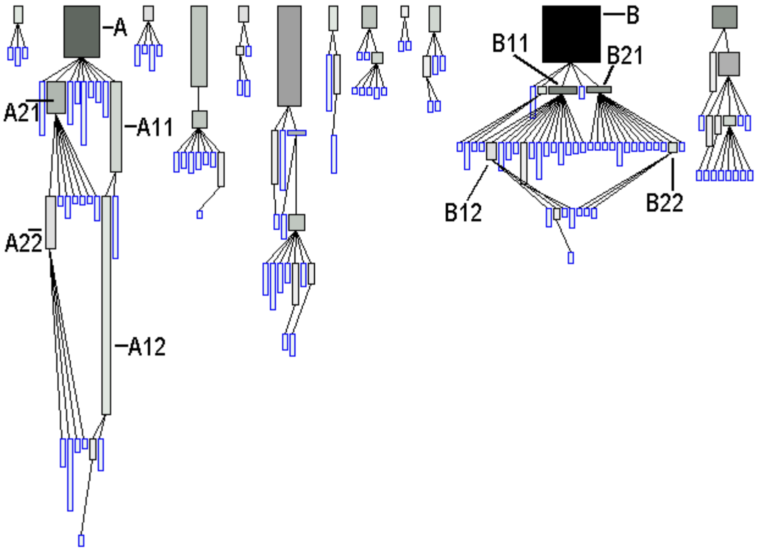


Figure 6.24: The spinoff hierarchy graph applied on the inheritance hierarchies of the Refactoring Browser. As size metrics we use WNOC and NOM, as color metric WNOC.

Results with Duploc: After removing the many stand-alone classes from Duploc, the remaining graph in Figure 6.25 can only show us the absence of spinoff hierarchies. Especially in the tree with the class *PresentationModelControllerState* (A) as root, we see that on the third level we have 5 siblings, 4 of which are all inheritance carriers, with only one tiny spinoff class with the meaningful name *PMCSDummyMode* (B).

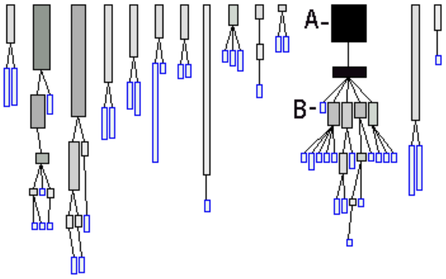


Figure 6.25: The spinoff hierarchy graph applied on Duploc. As size metrics we use WNOC and NOM, as color metric WNOC.

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

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

6.5.10 Inheritance Impact

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

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

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

Results with the Refactoring Browser: One of the hierarchies of the Refactoring Browser seems to have one such class which should certainly be further investigated: In Figure 6.26 we can detect the class *BrowserNavigator* (A) which implements many methods (175), while it only overrides one and extends two methods.

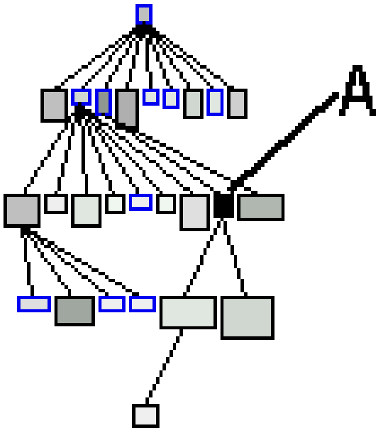


Figure 6.26: The inheritance impact graph applied on an inheritance tree of the Refactoring Browser. As size metrics we use NMO and NME, as color metric NOM.

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

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

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

6.5.11 Intermediate Abstract Class

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

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

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

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

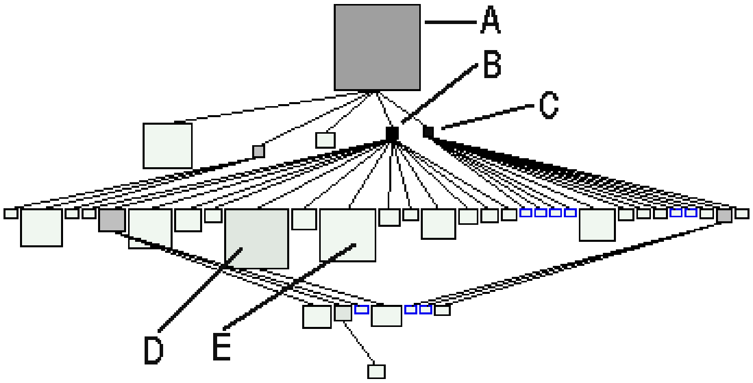


Figure 6.27: The intermediate abstract class graph applied on a subset of the Refactoring Browser. As size metrics we use NOM and NMA, as color metric NOC.

Results with the Refactoring Browser: The Refactoring Browser harbours in one of its inheritance hierarchies two intermediate abstract classes, as we see in Figure 6.27.

The root class *Refactoring* (A) implements quite a few methods, while we can spot the two intermediate abstract classes as *MethodRefactoring* (B) and *VariableRefactoring* (C). These two classes implement themselves very few methods (2 and 1 respectively) and are the roots of smaller subhierarchies. In the case of *MethodRefactoring* we see that its subclasses are implementing several methods, as we see in *InlineMethodRefactoring* (D) and *MoveMethodRefactoring* (E). Perhaps an attempt could be made to extract duplicated code and push it up into the intermediate abstract class.

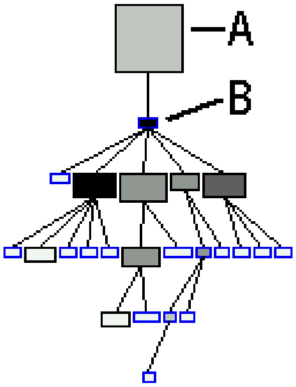


Figure 6.28: The intermediate abstract class graph applied on an inheritance hierarchy of Duploc. As size metrics we use NOM and NMA, as color metric NOC.

Results with Duploc: One of Duploc's inheritance hierarchies also contains an intermediate abstract class, as we see in Figure 6.28: The subclass *PMCS* (B) of the root class *PresentationModelControllerState* (A) implements only 4 methods and is obviously an intermediate abstract class. The subclasses of *PMCS* should be searched for duplicated code which could be pushed up into *PMCS*.

Possible Alternatives: None.

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

6.6 Method Graphs

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

In this section we list the following graphs:

- METHOD EFFICIENCY CORRELATION, Section 6.6.1.
- CODING IMPACT HISTOGRAM, Section 6.6.2.
- METHOD SIZE NESTING LEVEL, Section 6.6.3.

6.6.1 Method Efficiency Correlation

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

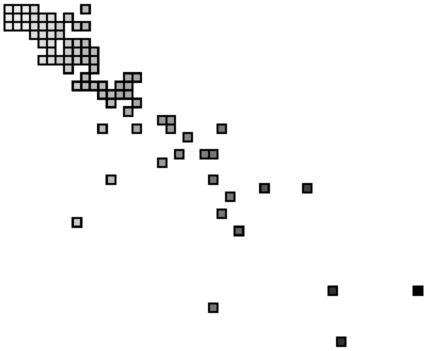


Figure 6.29: A method efficiency correlation graph.

General Idea: This graph is a good way to locate the *freaky entities* inside a group of methods, when it comes to their efficiency. By efficiency we mean how many statements are put on each line. By displaying the nodes in the correlation graph (as in Figure 6.29), we see that most of the nodes are near a certain correlation axis. However, there are a few which do not adhere to this rule.

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

1. High LOC (lines of code) and low NOS (number of statements). This is for example the case with ”forgotten methods”, that at some point have been commented out and then been forgotten. This may also be the case for overzealous line indentation, when a single parenthesis is put on a line of its own or when many blank lines have been used.

2. Low LOC and high NOS. This can be the case when the methods are written without indentation and several statements are on the same line, which is a bad thing too, since this decreases the readability, and it may also break the law of Demeter [Lieb 89].

3. Long methods (high LOC and high NOS). Normally a case for redesign, since long methods should be split up in smaller, better understandable and reusable ones [Beck 97].

4. Empty methods. These nodes position themselves on the top left of the graph. Although they can be viewed there by selecting and moving, the overlapping of

the nodes which is characteristic for this graph makes it hard to see those empty methods at one glance. A better graph for the detection of empty methods is the Coding Impact Histogram described in Section 6.6.2.

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

- Big nodes have many parameters. Although it's hard to define a threshold on the number of parameters, we think that methods taking more than 5 parameters should be looked at.
- Very small nodes on the outskirts of the graph should be looked at: these are very long methods which do not take any input parameter. Perhaps they could be split up easily.

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

Results with the Refactoring Browser: The method efficiency correlation graph shows some interesting results when applied to the Refactoring Browser. In Figure 6.30 we display all 2365 methods of the Refactoring Browser. We can spot several cases which should be looked into. The first nodes to meet the eye are those on the right edge of the graph (A). These three methods are very long (45, 51 and 65 lines of code) compared to the other methods in the system, which does not have a great distribution, thus signifying that the system is homogeneous related to the method lengths. The opposite case can be seen on the top left side of the graph (B). Upon closer inspection (by selecting and moving the nodes) we can see that the RefactoringBrowser contains 20 empty methods. The next point of interest is the method marked (C): this method takes 7 input parameters which is of course very much. The method *reInstallInterface* (D) on the top of the graph is also a case of closer study: While it has 16 lines of code it contains no statements. If we browse its source code, we see that the whole body of the method has been commented. The method *needsParenthesisFor:* (E) on the other hand contains 31 statements in only 19 lines of code and should perhaps be reformatted. The group of methods marked as (F) should also be looked into, since all of them contain comparatively few statements in long method bodies.

Results with Duploc: When this graph is applied to Duploc, as we see in Figure 6.31, the first thing to strike the eye is the large distribution of the nodes. Duploc obviously does have some very long methods. The second thing that meets the eye is that the main correlation axis has a different angle compared to the Refactoring Browser in Figure 6.30. The method *putPerlCode:* (A) is 201 lines long but does have only 2 statements. Upon closer inspection we see that its purpose is to print out a very long string. We have some other very long methods, (B) with 135 lines, (C) with 95 lines, (D) with 109 lines. We have some method that are far away from the system correlation axis, like (A), (C), (E), (F) and (G). (E) for example has 64 lines of code with only 1 statement. A closer inspection reveals its body is mainly commented code for

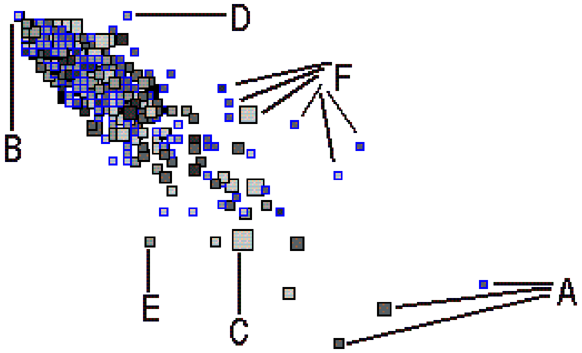


Figure 6.30: The method efficiency graph applied on the Refactoring Browser, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

testing purposes, i.e. when the system is tested some parts of the method body are uncommented. (F) reveals the same situation, where the 18 lines long method body doesn't contain any statements at all. (G) has 32 statements packed in 14 lines of code. Reformatting makes it more readable. The empty methods can of course be detected as (H), while we should also note the nodes around (I), which seem to be very short and at the same time badly formatted methods. The two methods (J) also draw attention due to their considerable size, which reflects the fact that they take 9 input parameters each.

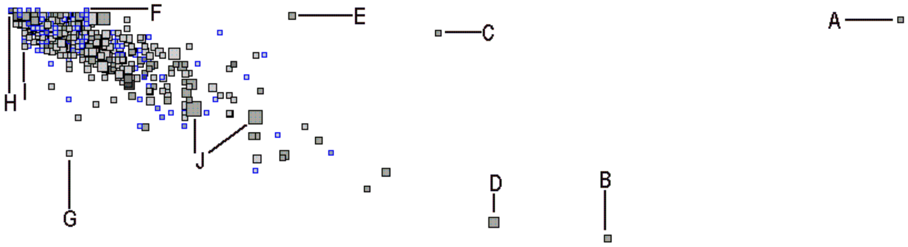


Figure 6.31: The method efficiency graph applied on Duploc, using as position metrics LOC and NOS, as color metric HNL, and as size metric NOP.

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

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

6.6.2 Coding Impact Histogram

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

General Idea: This graph shows the coding impact of methods and where the most coding has happened. While the normal histogram can only tell us how methods are distributed regarding their lines of code, this graph (Figure 6.32) can reveal where the real programming effort has been made: Writing 20 methods each one line long is easier than writing one method 20 lines of code long. It shows if there are any aberrant methods that are too long or if the system is unbalanced because of too long and complex methods. As a nice side-effect we can also grasp at one glance if there are any empty methods (those at the very top of the graph). A good design should have a lot of tiny methods so this is where the biggest columns in the graph should be. Methods not following this rule should be analysed as possible ”split candidates” which could be broken down into smaller pieces. While this graph is inefficient on whole systems because of the huge number of methods, it has proven to be very useful when applied to the methods of one single class. It should also be noted that the average length of a method implemented in typical industrial Smalltalk applications is around 6 lines [BECK 97].

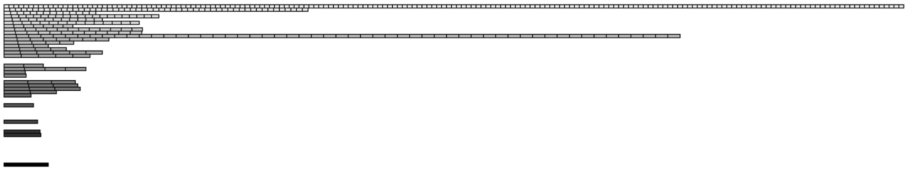


Figure 6.32: A coding impact histogram.

Results with the Refactoring Browser: Since this is one of the graphs which can hardly be applied on whole systems, but rather on specific small subsystems or singular classes, we do not compare the systems from our case studies with each other, but we rather show a few illustrative examples taken out randomly[7] from the Refactoring Browser. We selected only the two classes (*BrowserNavigator* (B) and *BRProgramNode* (A)) with the most methods for this graph. We see in Figure 6.33 that each class has its own coding impact topography. We see that *BrowserNavigator* (B) has many methods which tend to be overlong, and especially 6 very long ones which isolate themselves (B1) from the others. On the other hand *BRProgramNode* has an irregular topography with many accessors (A2) and one very long method (A1).

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

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

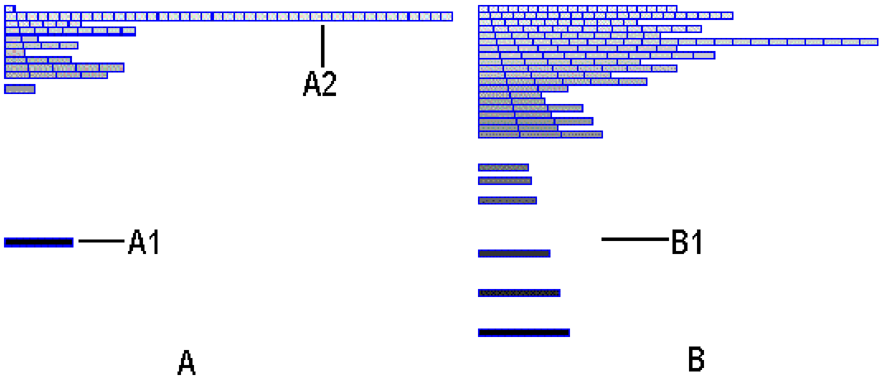


Figure 6.33: The coding impact graph applied on two classes of the Refactoring Browser. The width metric, as well as the color and vertical position metric is LOC.

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

6.6.3 Method Size Nesting Level

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

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

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

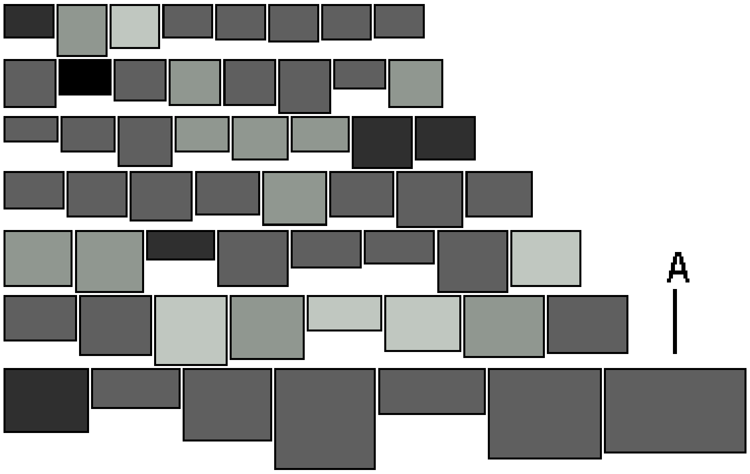


Figure 6.34: The method size nesting level graph applied on the largest Refactoring Browser methods. Size metrics: LOC, NOS. Color metric: MHNL.

Results with the Refactoring Browser: The Refactoring Browser shows in Figure 6.34 that is has been refactored itself a few times: there remain very few large methods, after filtering out all those with a LOC measurement smaller than 20. Yet, there are some large methods which also have medium MHNL values like those in the last row (A). Their lengths vary from 65 to 37 lines, which makes them also possible split-and-push-up candidates.

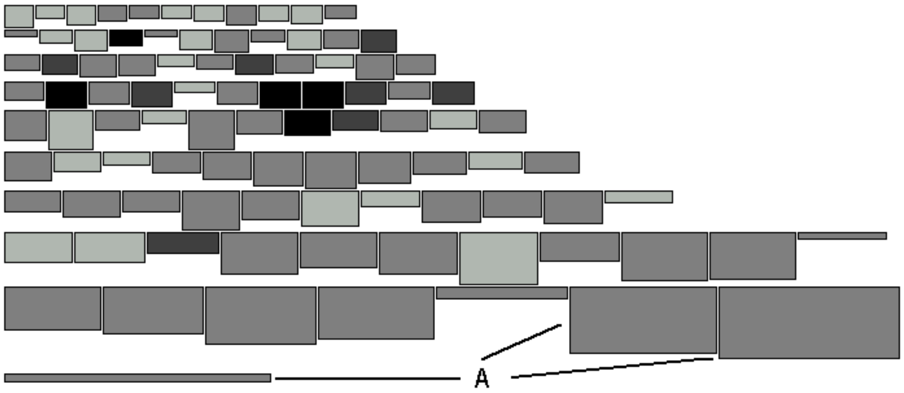


Figure 6.35: The method size nesting level graph applied on several Duploc methods. Size metrics: LOC, NOS. Color metric: MHNL.

Results with Duploc: We display in Figure 6.35 only the methods that have more than 20 LOC and belong to non-stand-alone classes. The resulting graph shows us there are several very large methods, which on one hand don't have big MHNL values, but since they're not methods belonging to root classes either, are all the same split-and-push-up candidates. The biggest methods (A) have 201, 135 and 109 LOC, which is way too much for Smalltalk methods. This excessive size is again due to the fact that most of them have never been refactored and written in one pull.

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

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

6.7 Attribute Graphs

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

In this section we list the following graphs:

- DIRECT ATTRIBUTE ACCESS, Section 6.7.1.
- ATTRIBUTE PRIVACY, Section 6.7.2.

6.7.1 Direct Attribute Access

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

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

Results with the Refactoring Browser: In Figure 6.36 we notice at once that there is the attribute *class* (A) defined in the class *MethodRefactoring* which is directly accessed 86 times. We also see there are some never accessed attributes which should also be further investigated (B).

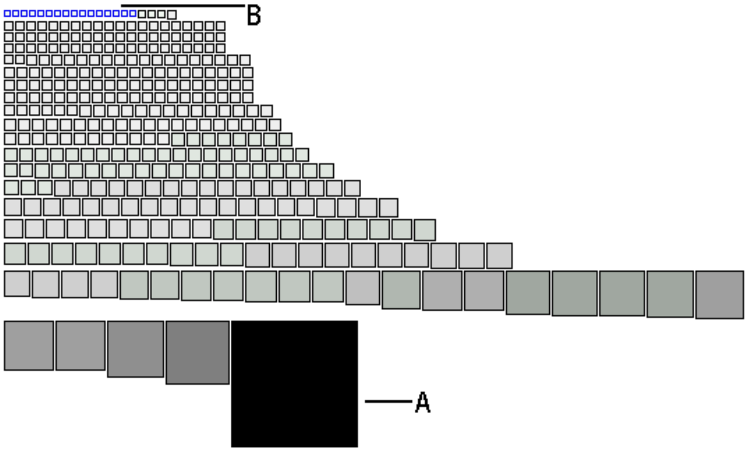


Figure 6.36: The direct attribute access graph applied on the Refactoring Browser. The size, color metric and sort criterion is NAA.

Results with Duploc: In Figure 6.37 we see that while in Duploc there are no attributes which are heavily accessed (the maximum is 31 direct accesses for the attribute *region* (A) defined in the class *AbstractRawSubMatrix*) there are many attributes which are never accessed (B) and which should be looked into for possible removal.

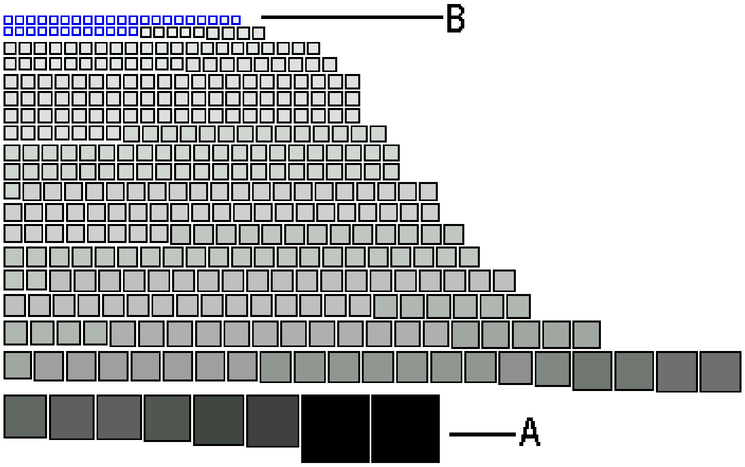


Figure 6.37: The direct attribute access graph applied on Duploc. The size, color metric and sort criterion is NAA.

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

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

6.7.2 Attribute Privacy

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

General Idea: Attributes may be directly accessed several times in a system. As we said in Section 6.7.1 such a situation is not ideal and can be detected with the graph described there. Apart from the number of times an attribute is accessed, another metric may prove to be useful for a similar graph: NCM, the number of classes which have methods that directly access a certain attribute. The attribute privacy graph is a checker graph which uses as size metrics NAA and NCM.

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

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

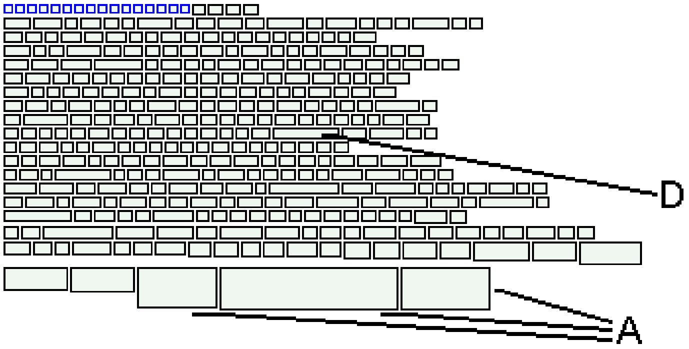


Figure 6.38: The direct attribute access graph applied on the Refactoring Browser. The size metrics are NAA and NCM.

Results with the Refactoring Browser: In Figure 6.38 we can spot some heavily accessed attributes marked as (A) which are accessed by many classes. We also see there are some very flat but wide nodes which are attributes heavily accessed by only 1 or very few classes.

Results with Duploc: In Figure 6.39 we can see that as a difference to the Refactoring Browser, Duploc has attributes which are seldom accessed by more than one class.

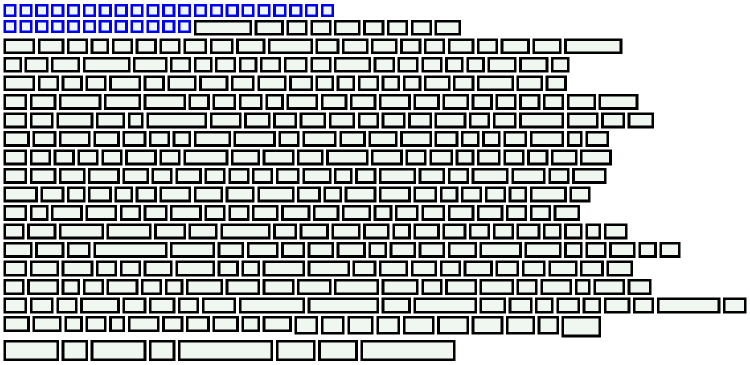


Figure 6.39: The direct attribute access graph applied on Duploc. The size metrics are NAA and NCM.

The maximum NCM value is 3. We deduce from that that the implementor of Duploc keeps an eye on encapsulation[8].

Possible Alternatives: None.

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

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

6.8 Class Internal Graphs

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

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

In this section we list the following graph:

- CLASS COHESION, Section 6.8.1.

6.8.1 Class Cohesion

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

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

Results with the Refactoring Browser: In Figure 6.40 we displayed the methods and attributes of the class *BRScanner* which has been identified as (C) in Figure 6.20. We gather at once that this class is heavily coupled internally and that splitting such a class is next to impossible.

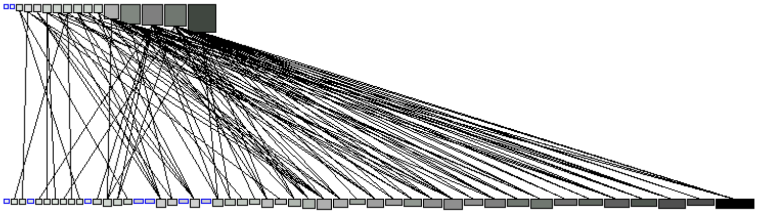


Figure 6.40: A class cohesion graph applied on the class BRScanner. The method nodes (in the lower row) use as size metric NOS and as color metric LOC. The attribute nodes (in the upper row) use as color and size metric NAA.

Results with Duploc: We obtained some impressive results when we applied this graph to some classes of Duploc. We show only one here: the class *DuplocApplication*. After filtering out all methods that never accessed attributes, we got the graph displayed in Figure 6.41 [9]. We clearly see two distinct clusters of attribute and method nodes.

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

This class is thus certainly a split candidate. This suspect was confirmed afterwards when I asked the implementor of Duploc about this class. He confirmed that this class was to be split up during the next redesign of the system.

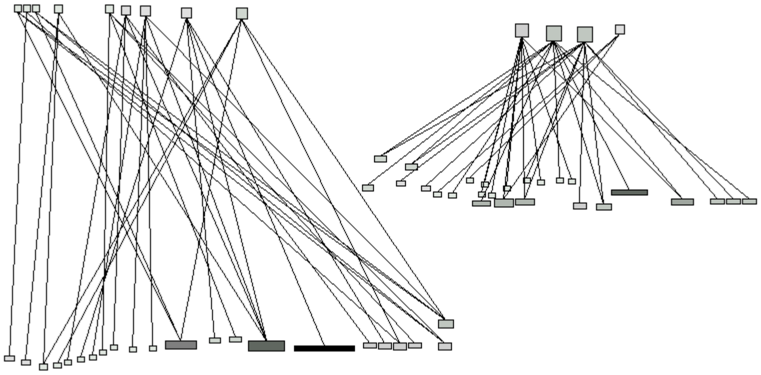


Figure 6.41: A class cohesion graph applied on the class DuplocApplication.

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

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

Chapter 7

Towards a Methodology

In this chapter we try to propose a methodology on how to reverse engineer a system with the approach discussed in this work.

In Chapter 6 we listed several useful graphs which can come in handy, but we did not always mention when and where they could be applied, and in which order. As they each address a different level of granularity, such a methodology is indeed important, because there is the risk to get lost in details if we start off in the wrong direction.

However, we emphasise again the playful nature of this approach, which can be summarised as *navigation through the code*. The fact that the graphs are interactive and that we can change the level of granularity at which we are moving through the system is primary. In no way can this methodology be seen as a strict set of instructions. It may very well be the case that we headed off in the wrong direction and tracing back our steps is often the only way out.

A good way to consider such a methodology is a labyrinth: Time and again we arrive at a certain position in the system where the path splits and we have several choices on how to go on. The more useful graphs we know, the better we can take our decision, because we have more choices at hand.

7.1 Getting an Overview

The first thing to do with an unknown system is to gain an overview. We should know how complex the system is (how many classes are there) and in which way the system is organised (how many inheritance hierarchies are there, how deep are they and is there multiple inheritance).

The first graph is thus a *system complexity* graph discussed in Section 6.5.1. This graph can answer us the above questions plus a few more: since it makes use of class size metrics like NOM, NIV and WNOC we are also able to detect extremely big classes at once. If the system is too big to fit on a single screen we could also select subparts of it and display it. Experience has also shown that removing all stand-alone classes from this graph can help reduce the complexity of the display, as in this case we're not really interested in the stand-alone classes. However, if one of the stand-alone classes is huge or many classes are very small or empty, we should focus our attention on those and after inspecting them come back to the system overview.

The overview part is characterised by this *hit and run* tactic: Each time we detect something interesting we head off in that direction and come back only when our curiosity is satisfied, while at the same time we have to take the risk of getting lost in details.

The system complexity graph has a drawback, which is the fact that the layout of the nodes depends solely on the layout algorithm. We will have a hard time at detecting very small or very big nodes if the graph is several screens wide. A *system hot spots* graph described in Section 6.5.2 alleviates this problem. Since the nodes should be sorted according to their size the detection of outliers gets easy. It also important there to play around with the metrics: The best strategy is to display the graph as described in Section 6.5.2 and change then the metrics following our interests.

The combination of these two graphs should be enough to get an overview. But we can use some other graphs to get further insights like *weight distribution* described in Section 6.5.3, *method protocol usage* described in Section 6.5.8, *root class detection correlation* described in Section 6.5.5, *service class detection* described in Section 6.5.6, etc.

The decision we have to take now is whether we focus ourselves on subparts of the system (especially inheritance hierarchies) or if we change the level of granularity (going down to methods, attributes and class internals).

A good way to postpone this decision is to display the whole system at a different level of granularity: We can display all methods and all attributes of the subject system with either correlation graphs, checker graphs or histograms.

At the granularity level of methods the *method efficiency* and the *coding impact* graphs described in Section 6.6.1 and Section 6.6.2 are certainly useful. Again, we stress the point of varying the metrics once those graphs are displayed to see what happens with the graph and to see if we can detect something interesting. One of the insights which could come from this is general information of how long the methods are and in which way they have been written, whether they are badly commented or not, etc.

At the granularity level of attributes we can use a *direct attribute access* graph described in Section 6.7.1 and again vary the metrics there. Possible insights could be how the attributes are used in a system and where they have been used, etc.

The best results we can obtain after getting an overview is a mental list of things we'd like to examine. This can be subparts of the system, inheritance hierarchies, big classes, small classes, certain very long methods, empty methods, attributes that are used very often or that are never used at all, etc.

With such results we can take the next step, which is diving into the system internals.

7.2 The Internals of a System

As internals of a system we consider to be subparts, sets of classes and their methods and attributes. It can even be just one class. The graphs that can be applied in this case can't be applied on the whole system without encountering some negative aspect like layout problems or graph size problems.

Note that every graph that can be applied to the whole system can be applied without problems to subparts of it. We won't mention them in this section to minimise redundancies in the text.

There are some graphs which can't be applied in certain cases, or where their application makes no sense: Every graph that deals with the inheritance aspect is useless if applied on single classes, etc.

The graphs that can be applied on inheritance hierarchies are the *spinoff hierarchy* described in Section 6.5.9, *inheritance impact* described in Section 6.5.10, *intermediate abstract class* described in Section 6.5.11, *method size nesting level* described in Section 6.6.3, *cohesion overview* described in Section 6.5.7, *attribute organisation* and *attribute privacy* described in Section 6.5.4 and Section 6.7.2.

Finally there are some graphs which can be applied only to single classes like the *class cohesion* graph described in Section 6.8.1 or which prove to be useful at this level of navigation, like the *coding impact* described in Section 6.6.2.

7.3 Scenarios of Navigation

It's hard to put up a real how-to, as there is no predefined path. Experience has shown however, that there are indeed some step-by-step techniques which are useful and which work most of the time.

An example. Suppose we want to look into a system to see if there are any split candidates. The first graph to be applied is the *system complexity* graph. If there are top-heavy hierarchies or big stand-alone nodes we should go on with a *cohesion overview* graph. If we are lucky we will detect some nodes which are worth a further investigation. We finally examine each node of interest with a *class cohesion* graph. Through interaction with the graph we possibly could end up with a split candidate layout. Once such a candidate is identified we have to look at the actual source code to really be sure that we have been successful with our navigation.

In the same as above several small navigation scenarios can be defined, each for a specific task, like splitting large methods, removing unused methods and attributes, pushing up or pulling down class functionality, etc.

7.4 Conclusion

The main lesson we can learn from this chapter is that the *navigational (lightweight) approach* discussed in this work is not a magic box, which can present meaningful results on a silver tablet.

On the contrary, this approach can only be really successful if it's used by an experienced reverse engineer who knows what he should look for and who can fully exploit the tool and its functionalities.

The advantage of this approach is its intuitive aspect, its high speed combined with an enormous reduction of complexity. The actual source code has to be read (it at all) only at the end of a navigation. This decreases the needed amount of time, cost and frustration of the reverse engineer.

Chapter 8

An industrial experience

We had the luck to test our approach on a real industrial case study during a five day workshop at the Nokia Research Center in Helsinki. The results obtained during the experience can't be directly discussed here, because of a non-disclosure agreement with Nokia. We want rather to present the knowledge gained during the workshop.

Context. The case study in question is a large application in the communication domain. Its size is approximately one million lines of code. It contains approximately 2300 classes and 25000 methods. It's implemented in C++ and C. It was thus very interesting to test our approach against language independency, scalability and platform independency.

The whole experience can be considered to be *real* in a wide sense: on the morning of the first day we didn't know anything about the case study, and we had to organize ourselves: install our tools, take a quick look at the case study, and apply our knowledge and tools on a large case study coming from a specific domain.

The main problems were indeed centered around limited resources: As one week is a very short amount of time for reverse engineering, it was not possible to study the application in detail.

The first two days were used to generate CDIF - files out of the C++ source code. This process is done by the tools Sniff+ and Sniff2Famix[1]. It worked all quite well, although we saw that the CDIF format has some holes which need to be filled. These holes are mainly present in the case of low-level programming[2].However, the fact that we succeeded in processing one million lines of C++ code into CDIF files showed the potential of the FAMIX model in an impressive way.

As the results had to be presented on the fifth day, for the actual experience only two days were left.

Goals. We set ourselves a list of goals we would try to reach during the workshop. The workshop was the first real acid test for CodeCrawler and our approach, because we had to reverse engineer under time pressure a very large and unknown system writ-

[1] To get additional information on this subject please consult the FAMOOS resources at *http://www.iam.unibe.ch/∼famoos/*.

[2] Two short examples: In the source code there were inheritance relationships based on conditional statements and unnamed structs. In the first case the CDIF parser interpreted it as a multiple inheritance, in the second case it gave the unannmed structs dummy names.

ten in a language which CodeCrawler could process only through the CDIF interface. We set ourselves the following goals for the workshop:

- Detect what graphs and metrics are really useful for such an experience. See if the graphs listed in Chapter 6 could be applied in all cases or if there were exceptions to their applicability.
- If possible, discover some new useful graphs which are specific for very large systems.
- Discover and remove bugs from CodeCrawler and the Moose framework.
- Study the way we were using our graphs repository and in which order. See if we could detect a methodology in the way we were approaching the system.
- Test our approach against scalability. See if the size of the application to be reverse engineered matters and in what respect it does that.

Results. During the workshop, we saw that nearly all of the goals listed above could were reached and we obtained the following results and insights:

- During the workshop we discovered some graphs which are especially useful for very large systems. A good example is the ROOT CLASS DETECTION graph described in Section 6.5.5: The subject system was so large that a inheritance tree graph would use several screens of space and looking for important root classes would be difficult and time consuming.
- Our approach is scalable indeed. Several times a simple visualisation could reduce the enormous complexity of the whole system down to an easily understandable graph, where hot spots or problem cases could be made out at once for further investigation.
- In Chapter 7 we point out a *top to bottom approach*. This worked well in this case: after a general overview on the whole system (mainly done by tree and checker graphs), we had to focus ourselves on interesting hot spots where we could reduce the level of granularity by diving into subsets of the system.
- A system which needs reengineering is seldom in a state in which it can be reverse engineered at once and without preparation. A mental preprocessing, consisting mainly of understanding the domain and purpose of the system must precede it. As we didn't have the time to obtain such information, we often had to proceed on assumptions about the actual functionality of classes. This was mainly done based on the class and method names and on the commentaries in the source code. Our knowledge of the system was very limited. It was very difficult to grasp a global overview of the system and to identify the roles of the different layers and subsystems. We could not easily identify the main applications or entry points. Tools can automate labour intensive tasks and can help in localising possible anomalies but expert judgements are still of vital importance for a reverse engineering approach. Although the source code was available, we could use it only for validating the results of our tools and for confirming our intuitions but not for getting more insights of the behaviour of the system. A next similar experimentation should be conducted or at least helped by a system expert.

- The fact that something looks suspicious on a graph is not enough to mark out spots which need reengineering. Real world programming often has to stick to time schedules and hardware requirements. Sometimes a quick and dirty solution which works, is far better than an elaborate solution which eats up time and money. Especially low level programming (which mainly addresses the needs of the underlying hardware) sticks to the quick and dirty rule. We saw that in the case of low level programmed parts of the system, the general rules of software design have to make place to hardware requirements. In such cases it's very hard to question the quality of the code.

- An insight that came up during the presentation of the results was that in some cases the programmers of the system were well aware of certain problems, but they let them stay because those parts of the system worked all right and needed no reengineering because their implementation was not to change anyway in the near future. Indeed, not every design problem needs a solution.

- As the subject system is implemented in C++, not all our metrics worked, because the first metrics we included in the Moose model were mainly Smalltalk and language independent metrics. During the workshop we implemented a few metrics which are specific for C++, but there are many more which could be added, as we point out in Chapter 9.

Conclusion. We were very satisfied with our tool CodeCrawler, as it worked well in every aspect. If we take stock of all those experiences, we can summarise it in the following way: Our approach is scalable and stable. We had a very positive feedback on CodeCrawler and proved its usefulness during the workshop. This acid test was also a good way for us to improve CodeCrawler and to lay the foundation of a methodology which we are trying to expand now. Right now we are looking for other industrial case studies on which we can apply CodeCrawler.

Chapter 9

Conclusion and Future Work

9.1 Summary

The intention of this work was to discuss a lightweight approach on object oriented software reverse engineering using *a combination of program visualisation, software metrics and interactivity*. We wanted to analyse how effective such an approach can be to help us to reverse engineer a software system and how scalable this approach is.

We began with a discussion of both fields:

- **Program visualisation** is already largely used in industry and is still a field of growing interest, as it has proven to be very useful to reduce complexity. The ways to visually display source code are diverse and in some cases very complex indeed. Our approach enables us to visualise up to five metrics at the same time for each displayed software entity. This is described in Chapter 4.

- We then focused our attention on the second area, **software metrics**. A constant point of discussion, the theoretical bases have been laid in the past few years and a great interest of ongoing research in this field is present. We use very simple metrics and list those in Chapter 3.

We then discussed a possible way to combine those two fields to use it for reverse engineering. We saw that there are some problems regarding the ways to visualise metrics, but at the same time we saw an appealing aspect of this approach, which is its intuitivity and flexibility.

We saw later on that combining those two fields with **interactivity**, we could extract several useful graphs which can be used for reverse engineering, and we laid the foundations for a methodology on how to approach systems for reverse engineering with this idea. We realize the concept of interactivity by using a graphics framework which enables us to dynamically change and interact with the displayed graph. We present our tool CodeCrawler in Chapter 5.

One of the points of interest is that with our idea we can get a different look at object oriented software: we are *navigating the code*. One of the prerequisites for this is that the actual source code is only mouse click away: we are not looking at static pictures, we are rather moving through the object oriented entities.

9.2 Main Contribution

The main contribution of this work is the following:

1. **Simplicity.** There is the appealing aspect of a lightweight approach. Without making use of complex algorithms or composed metrics we were able to obtain several useful results, which address either program understanding or problem detection.

2. **Scalability.** We saw that this approach is also scalable in many aspects, and that the enormous reduction of complexity is useful to reverse engineer even very large systems, as a one week workshop with an industrial case study clearly showed.

3. **Interactivity.** The interactivity of this approach showed a glimpse of a different way to look at software: the interactivity enables us to *navigate* through the source code and adds to this approach quite a playful and intuitive nature.

4. **Language and Platform Independency.** Our approach is language and platform independent. We stress this fact because this gives it an enormous flexibility and applicability. We tested our tool CodeCrawler on systems written in C++, Java and Smalltalk using CodeCrawler on the platforms Windows 95/98/NT, Unix, Mac OS and Linux.

9.3 Future Work

We pointed out on some occasions where possible extensions of this work could lie. This section summarises these extensions and presents additional ideas.

1. Extend CodeCrawler. First, the inclusion of additional layout algorithms which could increase the usefulness of our tool. Second, add more metrics which could be used by our tool. Third, extend the interactive potential of CodeCrawler, and add more functionality to it. CodeCrawler is right now a research tool, and there is a considerable risk that it will stay so. However, we decided to extend it in the future and make it available to the people. The incoming feedback would also help to decrease its flaws and increase its strengths.

2. Enlarge the repository of useful graphs we set up in Chapter 6. We are firmly convinced that many more useful graphs are waiting to be discovered, and in this context we're looking forward to applying our tool on other case studies, especially large industrial ones.

3. Such experiences could also help to increase the theoretical and practical aspects of an actual methodology. We started to lay the foundation of such a methodology in Chapter 7, and are convinced that there are great opportunities in this area. Another direction of study could be to make a direct mapping between such a methodology and the recently developed reverse engineering and reengineering patterns.

4. The underlying Moose model does not yet take data types into consideration. Smalltalk as such does not contain types. However, C++ and Java do, and the additional information which comes along with the types could be exploited to

increase the usefulness of our approach. The inclusion of data types into the Moose model is one of the next steps.

5. Right now the Moose model and CodeCrawler consider only three types of entities, namely classes, methods and attributes. There are more entities which could be considered as such: files, packages, applications and subapplications, directories, etc. The inclusion of such entities into the metamodel could also increase its usefulness and open up new paths of exploration.

6. The Moose model is currently supporting more than 40 metrics. There are many more which we want to add. In this regard the whole metrics framework of the model is soon to change to become more flexible and powerful.

7. CodeCrawler can right now be applied on static information. It would be interesting to see how such a concept can be applied on dynamic (run-time) information.

9.4 Final Remark

The importance of this work is reflected in the increasing importance of reverse engineering. Large object oriented legacy systems have become a major problem in the software industry. Their reengineering, which is preceded by a reverse engineering, has become a major economical factor in the software industry, and needs large amounts of time, money and human resources.

We are convinced that the ideas discussed in this work can be of great help to solving one of the largest problems present in software industry.

Appendix A

Graphs

A.1 Introduction

This chapter is dedicated to the graphs and layouts which were not used in Chapter 6. The discussion of the layouts is identical to the discussion of the graphs in Section 6.4.

A.2 The Circle Graph

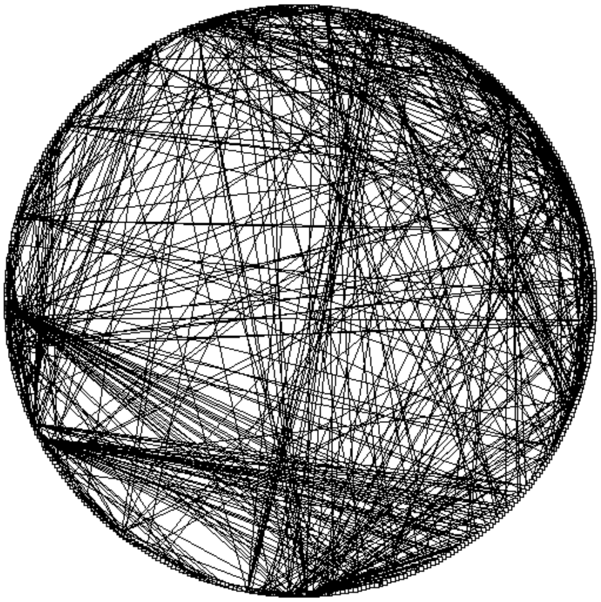


Figure A.1: A plain circle graph with method invocations.

Overall Idea. This idea originally comes from the field of psychology, where a circle graph is used to display relationships between people. We use it to display relationships between entities. However, the use is not advised for inheritance relationships between

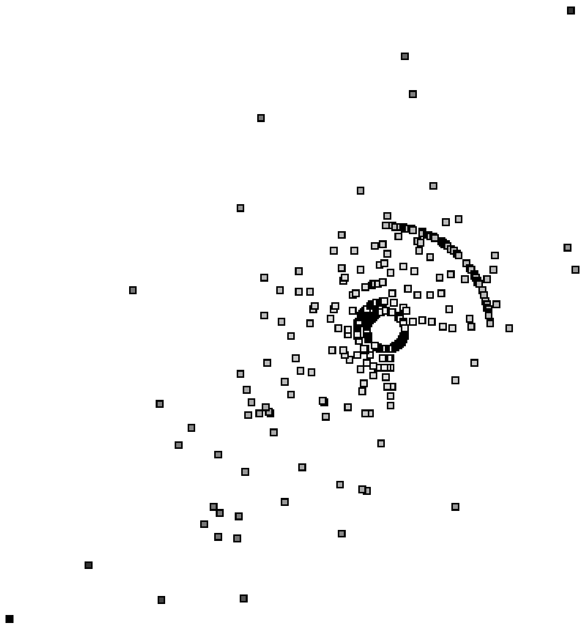


Figure A.2: A circle graph using a cloud layout.

classes, because those relationships are directed and because a tree graph can render such a situation much better. The best use that can be made with this graph is displaying invocation relationships between methods. The circle layout has a certain appeal because the eye tends to follow the imaginary circle line built out of nodes.

Scope. As we said, although it could be applied anywhere, we mainly use it to display methods, and especially all methods of one single class or of two classes invoking each other.

Layouts. There are several possible derivate layouts for this kind of graph. The original one is a plain circle with a fixed radius, which is given by the user. However, such a layout tends to get cluttered as soon as the number of nodes and edges grows. The first derivation is called *cloud*. A cloud graph displays each node at a certain distance from the center. The distance renders a supplemental position metric. The supplemental metric being its inherent advantage, there is also a certain drawback: since most metric measurements tend to be inside a certain interval with small values, the graph with a cloud layout looks cluttered around the center. We tried to remove this drawback with another derivate, called *spiral*. The only difference is in fact that the nodes are sorted according to the supplemental position metric. The nodes will most of the times be displayed as a spiral. The advantage that a spiral has, is that a greater number of nodes can be put on the same surface as a circle, and that since the eye tends to follow the lines of the spiral, the cluttering of the nodes will be alleviated. The next two layout derivates, called *concentric* and *inverse concentric* are very similar: the user has to define a number of layers. The nodes are positioned in one of those layers according to a position metric measurement. The advantage of the concentric layout is that a major number of nodes can be displayed on the same surface as a circle. A nice side effect

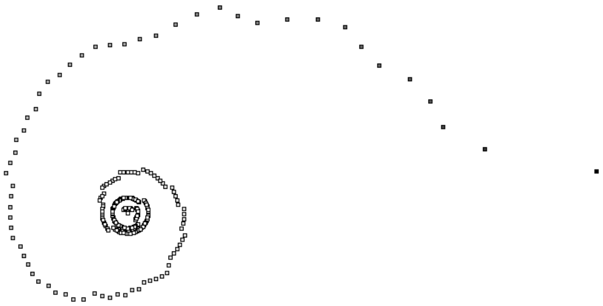


Figure A.3: A circle graph using a spiral layout.

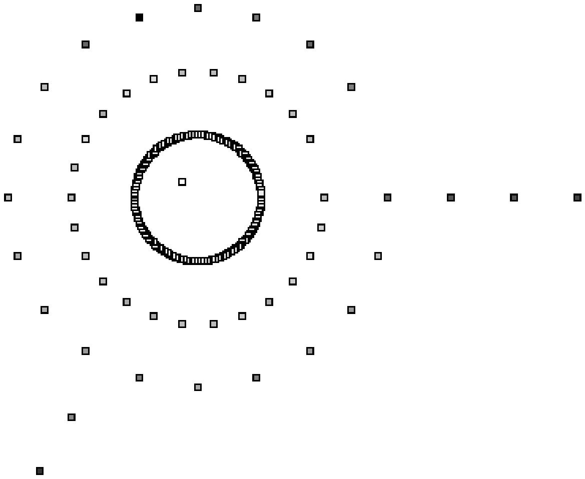


Figure A.4: A circle graph using a concentric layout.

is, that through the layer mapping we can classify the nodes in respect to their metric measurement. However, the concentric layout has a small drawback: Since most metric measurements are often small values, the concentric layout looks cluttered in the inner layers, while in the outer layers there will be only few nodes. In the inverse concentric layout we inverted the mapping function, so that the nodes with smaller position metric measurements are put in the outer layers of the graph. The only true drawback of the concentric and inverse concentric layout is that to be able to see on which layer the nodes reside, there has to be a certain amount of nodes on the layer to help the eye to mentally connect those nodes as belonging to the same layer.

Metrics. Each layout supports 4 metrics except the fixed radius circle layout which supports only three, namely size and color metrics. The cloud, spiral, concentric and inverse concentric layout support all a further position metrics which must be given by the user for the algorithm to work correctly.

Sort influence. A sorting of the nodes is effective in all layouts. In the case of the spiral layout it's even necessary for the algorithm to work, and in that case the sort

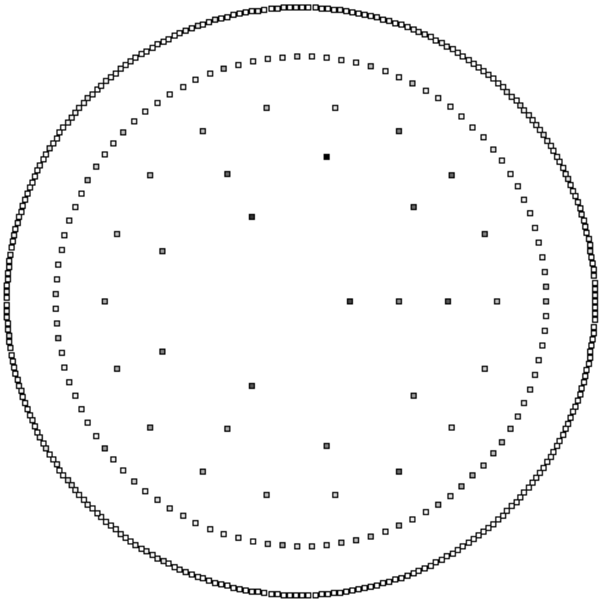


Figure A.5: A circle graph using an inverse concentric layout.

criterion must be the position metric. In all other layouts a sorting can sometimes help to make the graph less cluttered.

Pro et contra. The biggest contra for this graph is that a circle occupies quite a lot of space on the screen while it doesn't use it up well. We tried to minimise this drawback by derivating the initial layout. The pro is that the nodes are laid out in an appealing manner and if the number of nodes is small it is quite useful to get insights into classes. However, another major drawback is that the graph does not scale up well in case there are many nodes that have to be laid out.

Bibliography

[BAKE 92] B. S. Baker. *A Program for Identifying Duplicated Code*. Computing Science and Statistics, vol. 24, pages 49–57, 1992. (p 5)

[BALL 96] T. Ball and S. E. Eick. *Software Visualization in the Large*. IEEE Computer, pages 33–43, April 1996. (pp 1, 13, 14, 15, 16)

[BAXT 98] I. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. *Clone Detection Using Abstract Syntax Trees*. In Proceedings ICSM 1998, 1998. (p 5)

[BECK 94] K. Beck and R. Johnson. *Patterns Generate Architectures*. In M. Tokoro and R. Pareschi, editors, Proceedings ECOOP'94, LNCS 821, pages 139–149, Bologna, Italy, July 1994. Springer-Verlag. (p 28)

[BECK 97] K. Beck. Smalltalk Best Practice Patterns. Prentice-Hall, 1997. (pp 72, 76, 78)

[BRAN 95] J. M. Brant. Hotdraw. Master's thesis, University of Illinois, 1995. (pp 22, 28)

[BROO 75] F. P. Brooks. The Mythical Man-Month. Addison-Wesley, Reading, Mass., 1975. (pp 4, 5)

[CASA 98] E. Casais. *Re-Engineering Object-Oriented Legacy Systems*. JOOP, vol. 10, no. 8, pages 45–52, jan 1998. (pp 1, 5, 7)

[CHID 91] S. R. Chidamber and C. F. Kemerer. *Towards a Metrics Suite for Object Oriented Design*. In Proceedings OOPSLA '91, ACM SIGPLAN Notices, pages 197–211, November 1991. (p 9)

[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. (p 9)

[CHIK 90] E. J. Chikofsky and J. H. C. II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, pages 13–17, January 1990. (pp 4, 7)

[CHUR 95] N. I. Churcher and M. J. Shepperd. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, vol. 21, no. 3, pages 263–265, March 1995. (p 9)

[CONS 92] M. Consens, A. Mendelzon, and A. Ryman. *Visualizing and Querying Software Structures*. In Proc of the International Conference on Software Engineering, pages 138–156, 1992. (p 1)

[CROS 98] J. H. Cross II, T. D. Hendrix, L. A. Barowsky, and K. S. Mathias. *Scalable Visualizations to Support Reverse Engineering: A Framework for Evaluation*. In Proceedings of WCRE'98, pages 201–210. IEEE Computer Society, 1998. (p 1)

[DEME 99] S. Demeyer and S. Ducasse. *Metrics, Do They Really Help ?* In Proceedings LMO'99 (Languages et Modèles à Objets), pages 69–82. HERMES, Paris, 1999. (pp 1, 8)

[DEPA 93] W. DePauw, R. Helm, D. Kimelman, and J. Vlissides. *Visualizing the Behavior of Object-Oriented Systems*. In Proc OOPSLA '93, pages 326–337, October 1993. (pp 1, 13, 14)

[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 –pages yet unknown. IEEE, September 1999. (p 5)

[FENT 97] N. Fenton and S. L. Pfleeger. Software Metrics: A Rigorous and Practical Approach. International Thomson Computer Press, London, UK, Second edition, 1997. (pp 9, 10)

[FOOT 97] B. Foote and J. W. Yoder. *Big Ball of Mud*. In Proceedings of PLop'97, 1997. (p 5)

[HEND 96] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1996. (p 10)

[HITZ 95] M. Hitz and B. Montazeri. *Measure Coupling and Cohesion in Object-Oriented Systems*. Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95), October 1995. (p 9)

[JERD 97] D. Jerding and S. Rugaber. *Using Visualization for Architectural Localization and Extraction*. In Proc of Working Conference on Reverse Engineering, pages 56 – 65. IEEE Computer Society, 1997. (p 1)

[JOHN 92] R. E. Johnson. *Documenting Frameworks using Patterns*. In Proceedings OOPSLA '92, ACM SIGPLAN Notices, pages 63–76, October 1992. (p 28)

[KLEY 88] M. F. Kleyn and P. C. Gingrich. *GraphTrace – Understanding Object-Oriented Systems Using Concurrently Animated Views*. In Proceedings OOPSLA '88, ACM SIGPLAN Notices, pages 191–205, November 1988. (pp 1, 14)

[KONT 97] K. Kontogiannis. *Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics*. In I. Baxter, A. Quilici, and C. Verhoef, editors, Proceedings Fourth Working Conference on Reverse Engineering, pages 44 – 54. IEEE Computer Society, 1997. (p 1)

[LAMP 95] J. Lamping, R. Rao, and P. Pirolli. *A Focus + Context Technique Based on Hyperbolic Geometry for Visualiszing Larges Hierarchies*. In Proceeding of CHI'95, 1995. (pp 1, 14)

[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, LNCS 1543, pages 256–257, 1998. (p 1)

[LIEB 89] K. Lieberherr and I. Holland. *Assuring a Good Style for Object-Oriented Programs*. IEEE Software, pages 38–48, September 1989. (p 72)

[LORE 94] M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Approach. Prentice-Hall, 1994. (pp 1, 8, 9)

[MARI 98] R. Marinescu. *Using Object-Oriented Metrics for Automatic Design Flaws in Large Scale Systems*. In Object-Oriented Technology Ecoop'98 Workshop Reader, LNCS 1543, pages 252–253, 1998. (p 1)

[MOLL 93] K. Moller and D. Paulish. Software Metrics. IEEE Press + Champman & Hall, 1993. (p 9)

[MÜLL 86] H. Müller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986. (pp 1, 2)

[RIEG 98] M. Rieger and S. Ducasse. *Visual Detection of Duplicated Code*. In S. Demeyer and J. Bosch, editors, Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543, pages 75–76. Springer-Verlag, July 1998. (p 35)

[RIVA 98] C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. Master's thesis, Politecnico di Milano, Milan, 1998. (p 14)

[ROBE 97] D. Roberts, J. Brant, and R. E. Johnson. *A Refactoring Tool for Smalltalk*. Journal of Theory and Practice of Object Systems (TAPOS), vol. 3, no. 4, pages 253–263, 1997. (p 35)

[SAND 96] G. Sander. *Graph Layout for Applications in Compiler Construction*. Research report, Universitaet des Saarlandes, February 1996. (p 1)

[STOR 95] M.-A. D. Storey and H. A. Müller. *Manipulating and documenting software structures using SHriMP views*. In Proc of the International Conference on Software Maintenance, 1995. (pp 1, 2)

[SUGI 81] K. Sugiyama, S. Tagawa, and M. Toda. *Methods for Visual Understanding of Hierarchical System Structures*. IEEE Transactions on systems, man and cybernetics, vol. SMC-11, no. 2, February 1981. (p 1)

[TICH 98] S. Tichelaar and S. Demeyer. *An Exchange Model for Reengineering Tools*. In Object-Oriented Technology (ECOOP'98 Workshop Reader), LNCS 1543. Springer-Verlag, July 1998. (p 26)

[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. (p 5)