Grouping in Object-Oriented Reverse Engineering

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Daniele Talerico

2003

Leiter der Arbeit:
Michele Lanza
Prof. Dr. Stéphane Ducasse
Prof. Dr. Oscar Nierstrasz

Institut für Informatik und angewandte Mathematik

Abstract

Software Reengineering is a main issue in software industry. One of its main activities – *reverse engineering* – is concerned with trying to understand a software system and how it ticks. For the investigation and graphical representation of large and complex systems there are various tools for automated support. However, the information extraction with these tools is difficult, visualising certain aspects of a software system may overwhelm the observer with its complexity.

We discuss in this work an object-oriented reverse engineering approach using grouping. The intention of grouping is to create groups with components of a software system. The use of grouping has many benefits for a reverse engineer: it supports the program understanding and the design recovery, it adds higher abstraction levels to the system, and it permits to create different representations as well as alternate views of a system. Furthermore, the use of groups and the scalability of grouping are effective in reducing the complexity of large systems.

All the information we need to create groups is found in the software system. User-defined queries on the system perform the selection of components for the creation of new groups. We introduce the visualisation of groups and show their usefulness with a Smalltalk case study.

Acknowledgments

I dedicate this master's thesis to my parents who gave me life and never stopped supporting me.

Moreover, I would like to thank:

> First of all, the people who made this thesis possible: Michele Lanza and Prof. Dr. Stéphane Ducasse (the "Moose brothers"). Thanks for your precious feedback and support.
>
> My company Daedalos that has always supported my efforts related to my studies. In particular, I'd like to thank Thorsten, Marita, and the Swiss: Sibylle, Sander, Daniel, Juan-Carlos and Markus.
>
> The people I have known during my studies in Berne: Caló, Stefano (Dottó), Georges, David, Martig, the Scg people, and of course those who spent an infinite number of hours with me in the student's pool and the cafeteria, especially Tobias and Frank.
>
> Last but not least, my readers that spent their time by reviewing this document – namely Michele, Stéphane, and Prof. Dr. Oscar Nierstrasz. Furthermore, Sander, Gabriela, and Daniel. I always was grateful for your useful comments.

A big hug to my "dearest" brother Claudio. I'd like to know where I would be if you were not here. Of course, I will not forget Ursi . . . thank you for being there.

You always are with me: my best and most precious friends Juan, José (the compadres), Mighé, Roger(s), and Monica.

A big and tender kiss to Sandra . . . you gave me the power to fight and to finish my thesis. I will never forget that.

Contents

List of Figures

Chapter 1

Introduction

> *In the context software development, a legacy system is a piece of software that: (1) you have inherited and (2) is valuable for you. Moreover, legacy systems refer to systems that present the problems of aging software [Par94] [Cas98]: original developers no longer available, outdated development methods, monolithic systems . . . .* [Duc01]

The existence of *legacy systems* is typically attributed to the *software aging* process. Parnas [Par94] states that there are two types of aging: the consequence of failed modifications to a software system and the result of these changes.

> *Reengineering is defined as the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.* [CC90]

The goal of *reengineering* is to reduce the complexity of a legacy system sufficiently that it can continue to be used and adapted at an acceptable cost [SDN02]. The software re-engineering of legacy systems represents today a major concern in software industry [Tah00].

As we can see in Figure 1.1 reengineering can be separated into different stages, among these *reverse engineering*. Chikofsky and Cross [CC90] define reverse engineering as the process of analyzing a subject system to: (1) identify the system's components and their interrelationships and (2) create representations of the system in another form or at a higher level of abstraction. They also state that "the problems that re-engineers have to deal with are among other things the managing of the big amount of data and the complexity of the systems. A key to controlling these attributes is automated support". Todays engineers can make use of tools that help extracting the relevant information and that visualize the data.

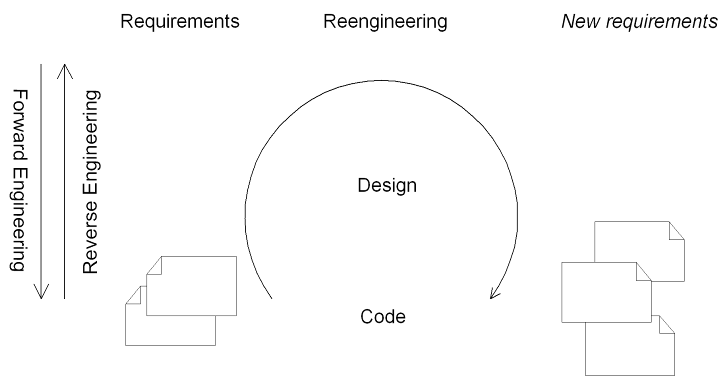


Figure 1.1: Forward, reverse, and reengineering.

There are many tools that make use of static information to visualize software. However, most publications and tools that address the problem of large-scale static software visualisation treat classes as the smallest unit in their visualisation [LD01]. Moreover, often they only focus on the logical components of a software system for graphical representation. In object-oriented programming these are typically classes, methods, attributes, or files[1].

There are mechanisms – for example, modules or inheritance with accessibility constraints – that can support layering or subsystem encapsulation, but they cannot enforce these abstractions [KC98]. Since abstractions are not logical components of a software system they are difficult to identify. Kazman and Carriere [KC98] state that tools alone cannot understand abstractions through which humans structure and communicate their systems. As an example they mention that subsystems are often determined by naming and calling conventions. Typically, automated aid does not provide a criteria-based grouping to identify such arbitrary conventions or abstractions.

[1]For many programming languages like C++ and Java the physical program structure is file-based.

1.1 Solution

In this master's thesis we discuss an object-oriented reverse engineering approach that uses a grouping method. It combines two aspects:

- logical grouping and
- graphical grouping.

The logical grouping is based on a software model[2], *i.e.*, we group the model's components. The model is an instance of the FAMIX meta-model.

With the graphical grouping we discuss the different possibilities for visualizing logical groups. On this occasion we introduce *CodeCrawler* [DDL99, Lan03], a research tool developed at the University of Berne. CodeCrawler provides us the necessary program visualisation support based on a software model.

1.2 Contribution

We reason about the systematic grouping in the reverse engineering process. The grouping involves adding higher abstraction levels to software systems. We show that with the creation of groups and their visualisation we can support the investigation and create alternate views of legacy systems.

We introduce a logical grouping based on a query-based selection of the components to be grouped. As grouping criteria we determine common properties of the components from a software model. We show that the model provides us all the information we need. Grouping is scalable and can be applied on all kind of components in the system. We show that grouping can reduce the complexity of the subject system.

We introduce a general description for group definitions. We select a collection of groups based on their impact on our problem solution and validate them for the understanding of software structures and the reverse engineering of software systems.

For the realisation of our approach we make use of CodeCrawler. We add more functionality and implement our features as an extension called *Classifier*. With the introduction of interactivity we want to provide a real user interaction with the model.

We detect the benefits of our approach and open the discussion for future work.

[2]The software model conforms to the FAMIX meta-model that we describe in section 2.3.1.

1.3 Document Structure

This document is structured in the following way. In Chapter 2 we present in detail the context of object-oriented reverse engineering. Chapter 3 outlines the solution to our reverse engineering approach. In Chapter 4 we provide a number of groups for the application on a meta-model. In particular we describe their definition and present the results acquired from the application on our case study. Finally, in the last chapter we discuss future work and improvements.

The appendix contains details about the implementation of our approach.

Chapter 2

Object-Oriented Reverse Engineering

The maintenance of legacy systems is difficult for many reasons. The reengineering process aims at the alteration of a software system in order to provide a more flexible system for maintenance and new changes. Reengineering generally includes some form of reverse engineering [CC90], the process that involves the examination and the abstraction of a system.

This chapter is organized as follows: in Section 2.1 we discuss the issues of legacy systems and software maintenance. In Section 2.2 we present the context of reengineering and reverse engineering. And we detect the key objectives of our reverse engineering approach. Section 2.3 introduces the FAMIX model, a language-independent meta-model that represents source code. This work is based on this model that was developed in relation with the FAMOOS project, described at the end of the chapter.

2.1 Introduction

2.1.1 Legacy Systems

> *In the context of software development, a legacy system is a piece of software that: (1) you have inherited and (2) is valuable for you. Moreover, legacy systems refer to systems that present the problems of aging software [Par94] [Cas98]: original developers no longer available, outdated development methods, monolithic systems . . .* [Duc01]

Legacy systems are typically mentioned in relation to *software aging*. Parnas [Par94] explains this phenomenon like this:

> *Software products do exhibit a phenomenon that closely resembles human aging. ... There are two, quite distinct, types of software aging. The first is caused by the failure of the product's owners to modify it to meet changing needs; the second is the result of the changes that are made.*

Typically, systems that are inherited and referred to the aging process are attributed to ...

- Growth: a) as software ages, it grows bigger due to the addition of new code [Par94]. b) at implementation time the developer team then gets split in many subteams.
- Unqualified authors that are faced with complex requirements.
- Error affected design that is reflected in the implementation.
- Error intrusion. As the software is maintained, errors are introduced [Par94].
- Missing program documentation ([SRT92]) or erroneous documentation because of iterative changes.
- Outdated development methods and tools.
- Unavailability of the original authors. Changes are made by people who do not understand the original design [Par94, SRT92].
- Modifications of the original design due to iterative changes to new requirements. After many changes the original designers no longer understand the product. The software becomes expensive to update [Par94].
- Unstructured programming methods [SRT92].

Now, because the software is *valuable* means that owners want to keep it and cannot just throw it away. In computer science we use for this the term *software maintenance.*

A legacy system can only be replaced or upgraded at a high cost. Replacing would mean implementing a new system from scratch and this would be too time consuming. The upgrading would be typically penalized by the problems mentioned above. In fact, the goal of *software reengineering* is to reduce the complexity of such systems to make them adaptable at an acceptable cost.

2.1.2 Software Maintenance

Maintenance of software becomes both more complex and more expensive [SRT92]. The ANSI definition of software maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to a changed environment".

Usually, the system's maintainers were not its designers [CC90]. This means that they first have to understand the system and recover its design. *Design recovery* is a subset of *reverse engineering*:

> *Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself.* [CC90]

To learn about legacy system and for the *program understanding* maintenance personnel must spend an inordinate amount of time attempting to create an abstract representation of the system's high-level functionality. They do that by expanding many resources and by exploring its low-level source code [SRT92]. Reverse-engineering tools can facilitate this practice [CC90]. In this context, reverse engineering is the part of the maintenance process that helps you understand the system so you can make appropriate changes [CC90].

2.2 Reengineering, Reverse Engineering

2.2.1 Terms and Definitions

Reengineering and reverse engineering are often mentioned in the same context. To understand the following terms and definitions we introduce a life-cycle model that is constituted of three simple stages (see Figure 2.1): requirements, design, and implementation. With this model the software reengineering and related terms can be described.

Now, here is the definition of *forward engineering*:

> *Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.* [CC90]

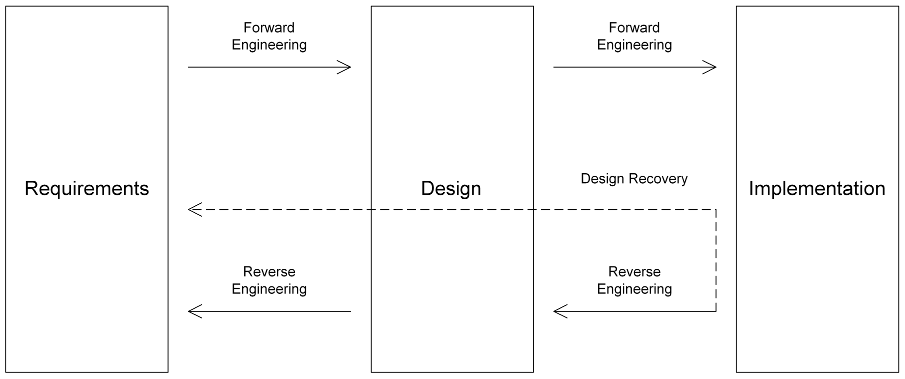


Figure 2.1: Life-cycle model. Reverse engineering and related processes in relation with the life-cycle phases of a system.

That is to say, forward engineering describes the common known process of developing a software system. According to the Figures 1.1 and 2.1 we can think of reverse engineering as the inverse procedure of forward engineering, namely extracting system abstractions and design information out of existing software systems for maintenance, re-engineering and reuse [SRT92]. Chikofsky and Cross [CC90] define it like this:

> *Software reverse engineering refers to the process of analyzing a subject system to*
>
> - *identify the systems components and their interrelationships and*
> - *create representations of the system in another form or at a higher level of abstraction.*

This definition states that the reverse engineering process does not modify a system. It is only focused on the examination of a subject system. It involves finding the original design and deciphering the requirements implemented by the system. Finally, we come to the definition of reengineering that includes some form of reverse engineering process [CC90]:

> *Reengineering...is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation in a new form.* [CC90]

As stated by the definition, reengineering consists of two main activities, namely the examination and the alteration of a subject system. More formal terms for these activities are reverse engineering and forward engineering [Tic01].

2.2.2 Objectives

There are many objectives in reverse engineering. Chikofsky *et al.* [CC90] concentrate them to 6 key objectives:

- Cope with complexity
- Generate alternate views
- Recover lost information
- Detect side effects
- Synthesize higher abstractions
- Facilitate reuse

In the following of this section we want to focus on the complexity, the views, and the abstractions. We then add the issue of *code smells*. In the following chapter we present a reverse engineering approach that pursues these objectives.

Cope with Complexity

For dealing with a huge amount of information one path that is followed these days, is the use of *software metrics*. The application of metrics is important and has been intensively discussed in the past [FP96, CK91, CK94]. Now, metrics have been developed further and today you can classify them in two groups according to Lorenz and Kidd [LK94].

- Design Metrics. These metrics are used to assess the size, quality, and the complexity of software. They measure 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.

- Project Metrics. They deal with the project's dynamics, with the overhead to get to a certain point in development cycle and the way how to get there. They can be used to predict for example staffing requirements. At a higher abstraction level they are less prescriptive and are more important from an overall project point of view.

Generate Alternate Views

A often used approach is the graphical representation of source code, namely the *program visualisation*. A graphical 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.

The visualisation we use for our reverse engineering approach is based on the Famix model, a meta-model for source code, that we present in section 2.3.

Synthesize Higher Abstractions

The goal of design recovery is to reconstruct the system design that preceded the first implementation. For this it is important to detect abstract system components at a higher level than the source code level. We also call these abstractions *modules*. It is useful for the design recovery to detect in a system the contained modules and their communication.

> *"Design recovery recreates design abstractions from combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains...Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth. Thus, it deals with a far wider range of information than found in conventional software-engineering representations or code."* Tedd Biggerstaff

Among the goals of the design phase in software development we find architecture design, particularly the modularisation of a system. In this context the term *module* represents a part of the system that is meant to be implemented independently. The modularisation of a system involves the creation of several modules. It is a characteristic that affects the quality of a program.

The most important information lies in the dependencies between these modules. These relationships can be detected by identifying the interfaces that provide their communication. As one possibility of our approach that we introduce in Chapter 3 we can create abstract components to identify modules.

Code Smells

The readability of program code is a characteristic for the software's quality. Well readable code is helpful for the examination of a software system. Programs that cannot satisfy this characteristic have a "bad smell". Martin Fowler and Kent Beck [FBB+99, Bec00] use the term *code smells*.

> *A code smell is a hint that something has gone wrong somewhere in your code. ... Note that a code smell is a hint that something might be wrong, not a certainty. ... Calling something a code smell is not an attack; it's simply a sign that a closer look is warranted.*
> (http://c2.com/cgi/wiki?CodeSmell)

Examples for code smells are:

- Duplicated code.
- Methods too big.
- Classes with too many instance variables.
- Classes with too much code.
- Strikingly similar subclasses.

When reengineering a software we try to eliminate code smells to give the system a better structure and to raise the readability. With this thesis we want to provide among other things a mechanism to detect code smells to point out possible spots for later refactorings.

2.3 FAMOOS

In Chapter 3 we present a reverse engineering approach that is based on a meta-model called the FAMIX model. It was developed in the context of the FAMOOS project. It basically represents source code. With MOOSE – the Smalltalk implementation of the FAMIX model – we are able to generate a model of a given software system to use it for the purpose to reverse engineer the system. In this section we give an introduction to FAMOOS and the FAMIX model. We explain MOOSE in more detail in the appendix of this document.

FAMOOS, the 'Framework based approach for mastering object oriented systems' refers to the european ESPRIT [1] Project 21975. The goal of the FAMOOS Esprit project was to support the evolution of first generation object-oriented software, built with current analysis and design methods and programming languages,

[1]ESPRIT projects are concerned with research and development activities in the area of information technology.

to frameworks – standard application architectures and component libraries which support the construction of numerous system variants in specific domains. Methods and tools developed to analyse and detect design problems with respect to flexibility in object-oriented legacy systems and to transform these systems efficiently into frameworks based on flexible architectures.

FAMOOS was a 3-year project that started at September 1996 and ended in September 1999. The project is set up with six partners: The Software Composition Group (SCG) within the University of Berne (UOB), Forschungszentrum Informatik (FZI), Daimler-Benz (DAB), Nokia Corporation (NOK), Sema Group (SEM), TakeFive Software (TAK).

For further information on FAMOOS refer to
`http://dis.sema.es/projects/FAMOOS/`.

2.3.1 FAMIX Model

The Famoos Information Exchange proposes a meta-model that can be compared to the UML meta-model [DDT99]. It was developed at the University of Berne[2] to offer a tool interoperability standard within the FAMOOS project. The preceding argumentation was that UML is not sufficient to serve as tool interoperability standard for integrating round-trip engineering tools. Under the features of the FAMIX meta-model the most relevant are:

- Language-independency. At the moment parsing technology exists to generate models from C++, Java, Smalltalk, and Ada.
- Software metrics support.
- Interoperability with program visualisation tools.

The meta-model comprises the main object-oriented concepts - namely Class, Method, Attribute and Inheritance Definition - plus the necessary associations between them - namely Invocation and Access. The core of the FAMIX meta-model is depicted in Figure 4.1.

A model represents a software system conforming to the FAMIX meta-model. Once a model has been built, we use it as a database of object-oriented entities. We can make queries to the model and its entities. Let us suppose we have an entity representing a class. We can now ask this class to give us all its methods, attributes, *etc.*

[2]See `http://www.iam.unibe.ch/~scg/Archive/famoos/FAMIX/` for additional information.

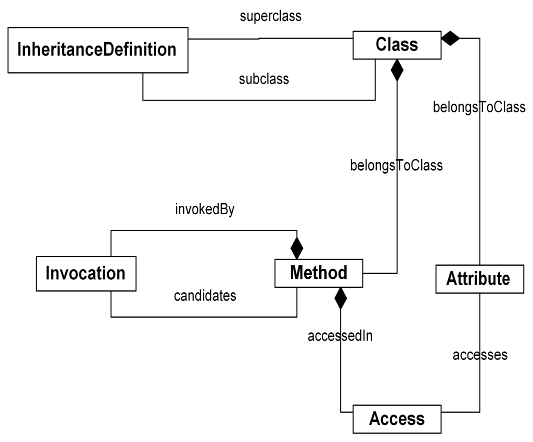


Figure 2.2: The core of FAMIX meta-model.

2.4 Summary and Next Steps

We start this chapter with discussing the issues in software industry: legacy systems and software maintenance. We present the context of reverse engineering and then expose the key objectives we want to pursue in our work. We finish with the introduction of the FAMOOS project and the FAMIX model.

In Chapter 3 we proceed with our reverse engineering approach. Our goal is to group arbitrarily the components of a software model. We investigate the difficulties and the prerequisites that grouping involves. The grouping has to be performed in two ways: logically and graphically. The logical creation of groups is focused on the FAMIX model. The graphical grouping instead is based on *CodeCrawler* (see Section 3.3.2), a reverse engineering tool that supports program visualisation.

In Chapter 4 we present a selection of groups for the investigation of legacy systems. We apply the group definitions on a concrete software system and discuss the results.

Chapter 3

Grouping

In the previous chapter we treated the context of software reengineering and discussed different objectives of reverse engineering. Finally, we introduced the FAMIX model – a meta-model for the practice in reverse engineering based on a software model. In this chapter we propose *grouping* as a reverse engineering approach. We want to group components of a software model that conforms to the FAMIX model.

This chapter has following structure: In Section 3.1 we introduce the context of our approach.

Grouping in reverse engineering basically consists of collecting components and relationships to abstract components. We call these abstractions *groups*. In Section 3.2 we propose an approach to define and use groups in the reverse engineering process. Afterwards we discuss how to represent groups visually.

MOOSE is the Smalltalk implementation of the FAMIX model. We would like to focus on two applications that were built on MOOSE: CodeCrawler and MooseQueries. CodeCrawler is a reverse engineering tool that supports program visualisation. MooseQueries provides an effective query framework for software models. Since the implementation of our approach makes explicit use of these applications we introduce them in Section 3.3.

In Section 3.4 we discuss the difficulties that grouping involves.

3.1 Introduction

The programming process involves moving from abstractions and designs to the physical implementation of a system [CC90]. Therefore a programmer has not only to write problem solving code but also to take part in the architecture and the design process of a software. As system requirements tend to change, existing code has to be modified and often the software's design has to be thought over. As Glass [Gla02] reminds us that for every 25% increase of problem complexity there is a 100% increase in complexity of the software solution. He also points out that explicit requirements explode by a factor of 50 or more into implicit design requirements as a software solution proceeds.

For a correct modification of a system we need good knowledge about the structure and the design. If the original developers are not anymore available we have to spend time examining the system to understand it. Automated support facilitates this practice and the handling of typically huge amounts of data. Since visualisation represents a good alternative to explore complex software systems several existing reverse engineering tools produce graphical representations of systems. The observation of their views is useful for the extraction and analysis of the components in the focused software. All the same we often wonder: but where does the information lie to adopt in a reverse engineering process? How can one decide what the right spots are in a system's view? How should one interpret a graphical representation? Why do reverse engineering tools often lack interactive features?

There are many tools that make use of static information to visualize software like Rigi [TWSM94], Hy+ [CM93] [CMR92], SeeSoft [ESEE92], Dali [KC99], ShrimpViews [SM95], TANGO [Sta90], as well as commercial tools like Imagix[1] to name but a few of the more prominent examples. However, most publications and tools that address the problem of large-scale static software visualisation treat classes as the smallest unit in their visualisation [LD01]. Table 3.1 lists the basic components of a object-oriented software program. Moreover, automated support only focusses on the logical components of a software system for graphical representation. In object-oriented programming these are typically classes, methods, attributes, or files. The standard views of a system used during design and implementation, such as the class and method structure as well as the source code, are often not sufficiently abstract [DD99]. Abstract views of a system are already supported in several design languages and methods. Examples include packages in UML [RJB99] or subjects in OOA [CY91]. Unfortunately, they are currently too vaguely defined, making them insufficient for the tasks occurring in reverse- and reengineering.

A visual representation of a code model usually provides interesting "attractive patterns". Especially when creating different views the observer recognizes clusters or *groups* within the software architecture created by the original designers. We consider these clusters abstract system components. Typically, we often recognize relationships between these components. The abstract components and their rela-

[1] See http://www.imagix.com.

tionships need to be analyzed. We want to know how they can affect the reverse engineering of the system.

The consideration of groups is not explicit in the software design approach or development.

> *...the abstractions live in the designer's head, or in usage conventions or naming conventions. These, however, are typically not automatically extractable by a tool: they are too idiosyncratic.* [KC98]

In this work we define and recognize abstract groups and their relationships in between. With the uses of extraction we want to understand software systems and we want to be able to propose refactorings.

3.2 Solution

In this section we propose our solution for grouping in reverse engineering. First, we list the requirements that our grouping concept should fulfill. After this we explain the grouping process. We split it in two parts: the logical grouping and the visualisation. At the end we discuss the behavior of our solution with respect to our requirements.

3.2.1 Requirements

For introducing groups into the reverse engineering process we propose a list of several requirements that have to be satisfied. We split them in two parts: grouping and application requirements. Note, if we use the term "system" we refer to a software model[2].

Grouping

Group Creation and Manipulation. We want to be able to create and manipulate groups for reverse engineering software systems. The group creation should be possible with the basic program elements, entities and associations[3]. Groups represent abstract components of a software system. For a flexible grouping groups should be applicable to an entire system and to subsystems.

[2]In particular our approach is based on MOOSE, the implementation of the FAMIX meta-model that we introduce in the following section. We presented the FAMIX meta-model in Section 2.3.1.

[3]Details are described in Section 3.2.2.

Group Concept Expressed as Queries. The selection of components for the grouping has to be expressed in the form of queries. This allows us to describe in a declarative way what properties our components must have. For example, we would like to "group all classes that have 10 methods and more".

Adding Abstraction Levels. Creating a group implies adding an abstraction level to the system. With the use of groups we want to describe abstractions like modules that are not concrete in source code. With the identification of abstract entities we want to be able to "think" in terms of class categories, packages, method protocols, *etc.*.

Code Smells. We want to detect code smells (see Section 2.2.2), *e.g.*, possible errors in the code or the original design.

Software Metrics. Our grouping approach should support software metrics for two reasons. First, the combination with the query concept enhances our grouping approach. With the use of metrics we can take more properties into account for describing the components we want to group. Second, relying on the selected components we can use their metrics to express metrics for the resulting group.

Lossless and Reversible Grouping. After the creation of a group, we want its items to be still contained in the original system. Similarly, a group's removal should not affect the system. The grouping of components must be reversible at any point of time.

Application

Customization. Our application has to provide a user interface for interactive group creation, storage, and visualisation. We want to provide a flexible handling of groups for the interaction with the system views.

Reusability. We want to have a group repository where we collect created groups. This allows us to adapt predefined groups to current situations.

Logical vs. Graphical Representation. Groups have always to exist in a logical form. From the visualisation point of view the observer should be able to choose between several representation forms since it is an objective of the reverse engineering process [CC90].

Put together, these requirements already set up a kind of an informal definition or concept of grouping. In the following section we want to give a more precise description of the grouping process.

3.2.2 Logical Grouping

Our approach is based on the FAMIX model that we introduced in Section 2.3.1. MOOSE is the Smalltalk implementation of the FAMIX model. In our implementation MOOSE provides us a software model. We treat MOOSE in more detail in Section 3.3.1.

The following terms are explained in more detail in the next chapter, see Section 4.1. Anyhow, we give here a short description for the understanding of the grouping process.

We use the term *group* in the meaning of abstract *entity* or *association* in the context of the FAMIX model. Table 3.1 describes the core structure of the FAMIX model. A group basically contains a collection of entities respectively associations as well as nested groups.

Atom	Description
Entity	Class, method, or attribute
Association	Class inheritance, method invocation, or attribute access

Table 3.1: The FAMIX core distinguishes between two types of components: entities and associations.

Here, we propose an informal definition for groups respectively grouping:

- A *group* is an abstract entity that contains a subset of all entities in the focused system. A group containing one entity is the entity itself.

- The logical grouping consists of creating a *group* with a set of entities. The structure of a group is illustrated by Figure 3.1.

  The grouping process can be divided into two parts: the entity selection and the group creation. We perform the entity selection with the execution of a query on the software model. The result consists of a set of entities. A group is created with the addition of a set of entities to an empty group.

- A grouping can be defined by a query that basically consists of selecting a set of entities in a software model.

The grouping in relation with associations is analogous.

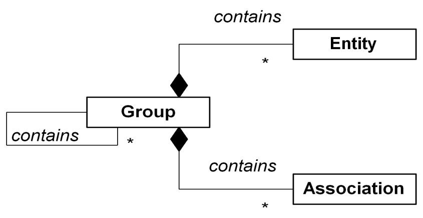


Figure 3.1: A group can contain either entities or associations.

3.2.3 Visualisation

Most of our work is based on CodeCrawler (see Section 3.3.2) – a reverse engineering tool that supports program visualisation. In the following of this section we show the outlines CodeCrawler as an example for a common reverse engineering tool.

The visualisation of a group can be performed in different ways. To understand this section we have to know how CodeCrawler displays software model's components graphically. CodeCrawler knows two types of graphical atoms:

- *nodes*: represent logical entities like classes or class methods.
- *edges*: represent relationships respectively associations, *e.g.*, inheritance, access, ... *etc.*.

Nodes represent entities, edges represent relationships or associations – we explain these terms in more detail in section 4.1. Figure 3.2 illustrates the basic graphical atoms of CodeCrawler. CodeCrawler creates the nodes and edges according to the entities and associations in the software model. After creation it stores the components in a data structure called graph.

Visualising a Group

For the graphical grouping we propose three different methods:

- Color/Selection. In this visualisation mode we pick all items belonging to a group and select or color them to be distinguished from all other nodes in the same view. Selecting a node means marking it with a little black rectangle in

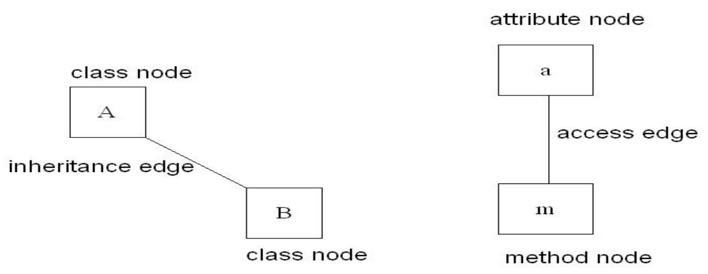


Figure 3.2: Entities are displayed as nodes. Relationships between entities correspond to edges.

the middle. Coloring a node implies filling a node with a determinate color. Both coloring and selection are illustrated in Figure 3.3.

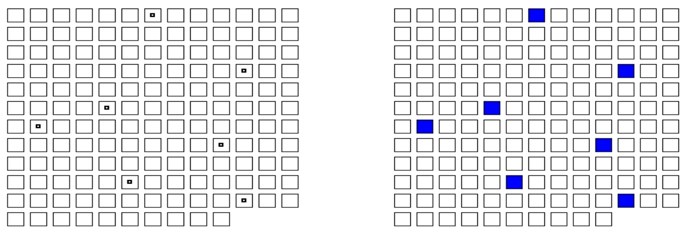


Figure 3.3: On the left all selected nodes belong to a group. On the right the group's item are colored blue.

- Aggregation. It consists of taking 2 or more nodes and substituting them with a new node, a *composite node*. Inside the node the logically contained nodes are visible. Figure 3.4 shows a typical node aggregation. The original connections (edges) of the aggregated nodes are preserved. If the composite node is connected with another node by two or more edges they get substituted by a composite edge. Similarly, the composite edges contain the original grouped edges. Note that for edge grouping we do not make use of aggregation but of collapsing.

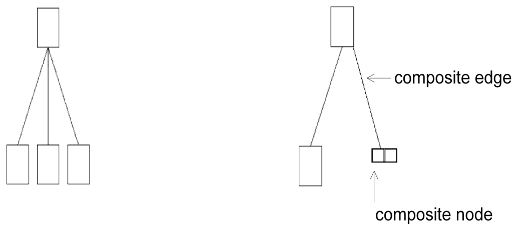


Figure 3.4: A node aggregation: the aggregated nodes are visible within the composite node.

- Collapsing. Here the same thing happens as in the aggregation with the difference that the contained items are not visible, see Figure 3.5.

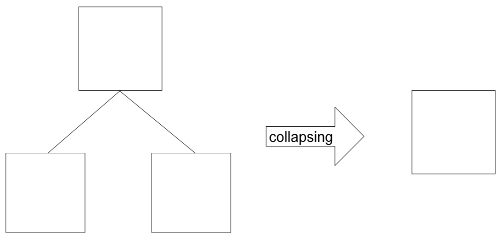


Figure 3.5: A node collapsing: the collapsed node hides visually its items.

For the visual grouping based on edges we only focus on the collapsing. This is because we have decided to put more effort in the analysis of the entity grouping. Entities are visualised by nodes. As we explained above edges represent relationships.

3.2.4 Synthesis

In this section we note how our solution behaves with the requirements listed above.

Group Creation and Manipulation. This is straightforward.

Group Concept Expressed as Queries. This is done with the use of Moose-Queries – this is an application we introduce in the following section.

Adding Abstraction Levels. This is straightforward. The creation of a group creates an abstract component.

Code Smells. With use of query-based grouping we can define groups based on specific code smell "patterns".

Software Metrics. CodeCrawler treats a system's components as nodes and edges. These are connected with software metrics in order that they are anytime available. CodeCrawler typically uses metrics for the visual enhancement of the components, *e.g.*, it determines the components' color and size in dependency of their metric values.

Lossless and Reversible Grouping Grouping involves only one system modification: after creation it adds the group to the system. After removal the original items remain in the system.

The logical reversibility is ensured during the deletion of groups. Since groups only hold pointers to their components, the model's entities will never be affected by a deletion. The reversibility is more interesting on visualisation side since expansion of groups restores the previous view.

Customization. Our implementation provides a graphical user interfaces called *GroupGenerator* for both the creation of groups and their graphical handling.

Reusability. With our application every created group gets stored in the repository. The repository allows one to remove groups.

Logical vs. Graphical Representation. This is straightforward. We explain the two concepts in Section 3.2.2 respectively in Section 3.2.3.

3.3 Constraints

For our implementation we set some constraints. We want to base our work on CodeCrawler, a reverse engineering tool that provides program visualisation and graph manipulation. The visualisation is based on a software model. The generation of the model is done by MOOSE, the implementation of the FAMIX model.

3.3.1 Moose

Moose is the implementation of the FAMIX meta-model that we presented in Section 2.3.1. It has been developed in Smalltalk by S.Demeyer and S.Ducasse, at the

University of Berne. Starting from a given system, implemented with a object-oriented programming language like Smalltalk, it produces a model. If the software has been developed in any other language than Smalltalk we make use of particular exchange format called CDIF. In this sense the model is language-independent. We can query its entities and associations, *e.g.*, an entity mapped as a class can return its methods or answer if it is abstract or not.

The data model contains the main object-oriented concepts, namely *class, method, attribute* and the associations as *inheritance definition, invocation and access*. We explain this terms in more detail in Section 4.1.

Moose produces the main knowledge base, CodeCrawler is based on the graphical representation of the FAMIX model.

MooseQueries

This application extends the Moose framework. Its main contribution is a querying interface to the FAMIX model. The interface allows one to express queries for the model's entities and associations. MooseQueries was written by Lukas Steiger at the University of Berne, in the context of his master thesis. For further information see [Ste01].

3.3.2 CodeCrawler

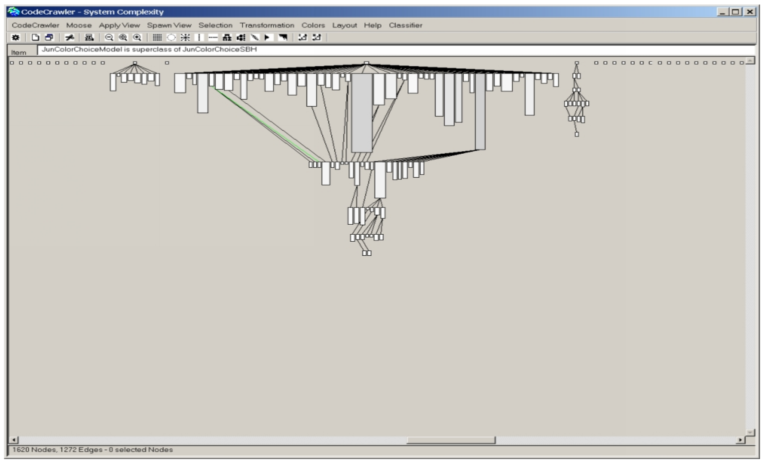
This program was built on top of MOOSE and supports the visualisation of a software model. In particular it supports:

Generation of Graphical Views with different perspectives on software architectures, *e.g.*, class level or method level, for the analysis and understanding of software systems. The application provides a long list of predefined views. Every view can be represented with several layouts.

Metrics-based Visualisation. CodeCrawler uses a number of operators that compute metric values. This values are combined with the visualisation of the nodes and edges changing that way there size and color.

CodeCrawler was developed with VisualWorks Smalltalk and built using the framework HotDraw[4]. The platform independence is given by the programming language Smalltalk since it is based on a virtual machine.

[4]HotDraw is a two-dimensional graphics framework for structured drawing editors that was implemented in Smalltalk by John Michael Brant [Bra95].


CodeCrawler - System Complexity
CodeCrawler Moose Apply View Spawn View Selection Transformation Colors Layout Help Classifier
Item
JunColorChoiceModel is superclass of JunColorChoiceSBH

1620 Nodes, 1272 Edges - 0 selected Nodes

Figure 3.6: CodeCrawler, a reverse engineering tool that supports program visualisation.

3.4 Difficulties And Drawbacks

There are a few points that have to be considered when using grouping in reverse engineering:

Query Capabilities. The main feature of a query-based grouping is a flexible system interface for the access of the static information that lies in the source code. Moreover, it is important when coping with large systems that the component properties can be accessed in short time. Usually, a query addresses all system components.

Component Properties. The properties represent the most powerful element of query-based grouping. That definition of groups is totally based on properties. The underlying model has to provide a large set of properties for every entity and association. MOOSE provides several operators[5] that compute values for properties like a class type, a metric, *etc.*.

Homogeneous vs. Heterogeneous Groups. A group functions as a container. From this point of view it is either an entity group or a relationship group. We decide – for the logical and the graphical aspect – that there will be only entity and relationship groups.

[5]An operator is essentially an automated procedure that iterates over every component of a system to retrieve or calculate certain values.

Metrics Handling. The calculation of metric values for a group is tricky because it is based on its items. Ideally, the solution is a function of the group's elements. In this work we sum all values if the metrics are numeric. With properties we consider `true` if all items return true concerning the correspondent property.

Common Level of Granularity. The usage of groups introduces a powerful handling of a software model. The resulting system views change visibly without substantially modifying the model itself. There is the risk of creating groups representing different abstraction levels in the same view. This can distort the view's interpretation if the observer is not aware of the situation.

Data Size. For the reverse engineering of large systems we usually rely on automated tools. Big volumes of data affect considerably the tools' time performance. For example, the creation of a group can last several minutes with a software model containing a thousand classes and thousands of methods. For the creation of a graph CodeCrawler can take several hours.

Graphical Edge Representation. Edge visualisation is a tricky and time consuming task. One question we have to ask ourselves is whether we want to show or not the edges between a couple of nodes that we aggregate in a larger composite node, as we can see in Figure 3.4. The figure also illustrates that we collect all edges between a collapsed or aggregated node and another node in a single composite node.

Chapter 4

Groups

In Section 3.2 we introduce in detail the grouping process. We give an informal definition of the logical grouping and show how groups can be represented graphically. In this chapter we show what impact grouping in reverse engineering can have. For this purpose we present a list of groups that we have selected during the work done for this document. The groups are represented in *CodeCrawler*, a reverse engineering tool [Lan99].

This chapter is structured as follows: In Section 4.2 we show the general structure of the groups that are presented later in the chapter. With Section 4.3 we describe the different kinds of groups we have created and that we want to investigate.

We call the implementation of our approach *Classifier*. To test Classifier and to show the results of grouping on a concrete software system we chose a software called *Jun*. Jun is a freely available graphic library. We introduce it briefly in Section 4.4.

The main contribution of this chapter lies in Section 4.5. There we list in detail the most useful groups that were raised during this research. For every group we explain our intent and show the results for our case study.

4.1 Introduction

Our work in this chapter is based on the following definitions. The basic components of a software model conform to the FAMIX model are either *entities* or relationships, the *associations*. Figure 4.1 shows how entities and associations are connected with each other, Figure 4.2 shows how they are modeled and inherited.

We see in Figure 3.1 (page 20) that a group, or *computedGroup*, basically contains a set of entities or associations.

For more detailed information on the FAMIX meta-model please consult the PhD thesis of Sander Tichelaar [Tic01].

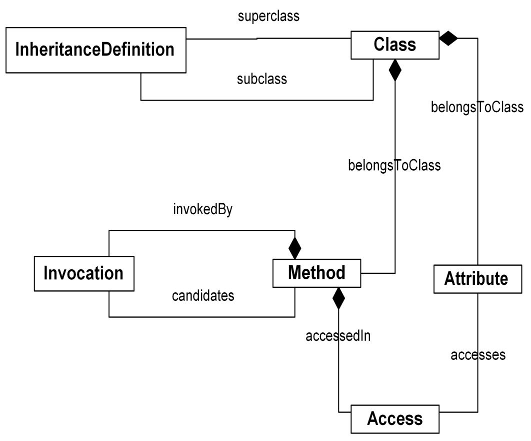


Figure 4.1: The core of FAMIX model.

We would like to stress the fact that in the following sections we use the term *system* in the meaning of software model.

4.2 Group Template

The following description is closely related to the graphical representation of a group. It illustrates the format we use in this chapter to discuss the groups we present in Section 4.5.

A group is defined as follows:

Values

- Components. As components one can chose entities from the metamodel. As we mentioned in Section 3.2.3, in this work we make use of class nodes and method nodes because we think that their grouping results are the most expressive for the discussion in this work.
- Selection criteria. This stands for the values that the group's components have to satisfy. For the software metrics deployed we used the abbreviations defined in Chapter 3 of *Combining Metrics and Graphs for Object*

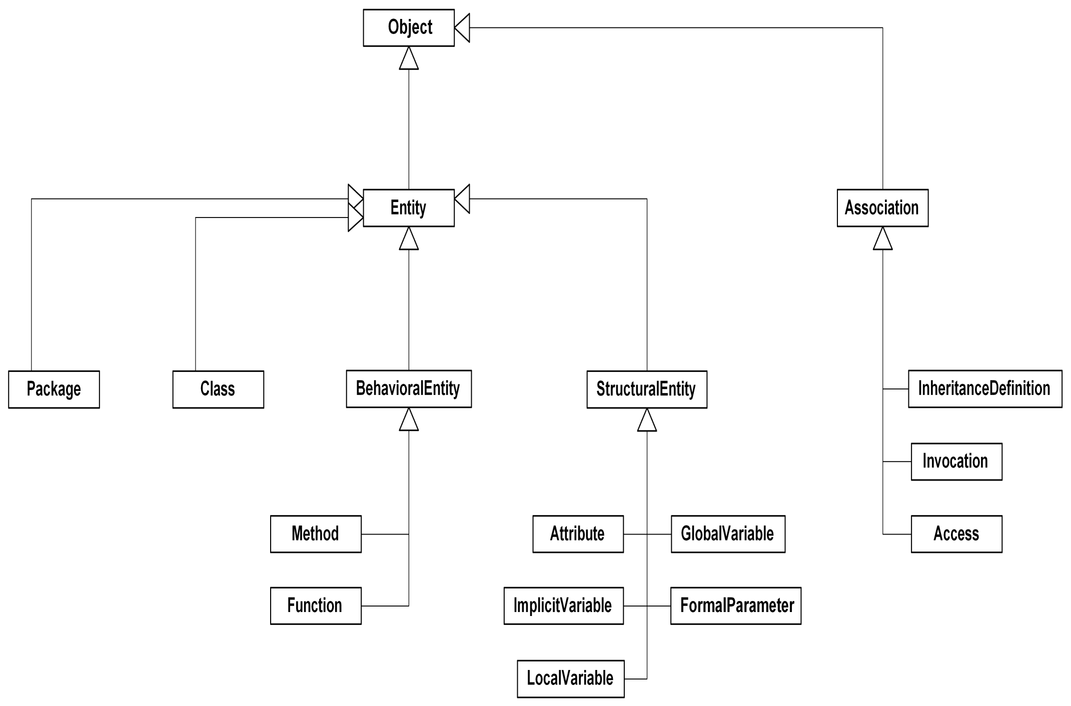


Figure 4.2: Entity and association within the FAMIX hierarchy.

Oriented Reverse Engineering from Michele Lanza [Lan99]. Tables 4.2 and 4.2 list the metrics used in his work.

If we want to use negated criteria we write ¬ before each criterion. Here is an example: $\neg isAbstract$ means *is not abstract*.

- Edges. Relationships are displayed as edges. Here we determine how the node entities have to be connected, see also Figure 3.2.

  In the case we write *arbitrary* we mean that the edge type is not important for the discussion of the group.

- Display mode. The display mode stands for visualisation mode. It expresses the way the groups' items should be marked. The value for the display mode is either *color/selection*, *collapsing*, or *aggregation*. These values were explained in section 3.2.3.

  In the case we write *arbitrary* we think that all three alternatives are appropriate.

Intent. We describe here what one can expect as result. What will the group look like, how are its components distributed all over the system. Additionally we give some hints, where appropriate, how to vary the input data to change the results. We write down, when possible, what problems could be isolated with the results.

Case Study. Here one can read the results obtained after the group's application to the case study that we present later on.

Interpretation. This point clarifies how the results can be interpreted. For instance, we propose some hints for refactorings, or we make emphasis on detected problems. We want to provide some support for reverse engineering and reengineering.

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 4.1: Class metrics.

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 4.2: Method metrics.

4.3 Kinds of Groups

For grouping in reverse engineering we propose two kinds of groups: simple and composed groups. Simple groups have only one selection criterion or just a few criteria. They serve as base for the extension to more complex groups with more detailed information. In the latter case we speak about *composed groups*.

Composed groups are basically the result of refining simple groups.

4.4 Case Study

As case study we chose the 3D Graphic Library called *Jun*. Jun for Smalltalk is a library that has been developed to bind OpenGL and QuickTime to VisualWorks, and for implementation of fast 3D graphics and multi-media application. It handles both topology and geometry, also supports handling multi-media data formats such as movies or sounds. Currently the package is being implemented in VisualWorks (Smalltalk), but a parallel project to implement it in Java has been started as well. SRA (Software Research Associates, Inc.) is distributing Jun with all source code at no cost under the terms of the GNU General Public License.

We analyzed version 398, which consists of more than 600 classes, 15'000 methods, and 2'000 attributes.

The interesting aspect of Jun is its variety as it models a wide spectrum of domains: different format readers and writers, different composite structures (HTML, VRML), various complex rendering algorithms, even a Prolog interpreter and a Lisp compiler and interpreter. Jun is mature and professionally developed. Currently, Jun is being used as the development tool for the Static/Dynamic Shape Conformation Evaluation Platform being developed under contract from the aggregate corporations Research Institute of Human Engineering for Quality Life (HQL) and National Institute of Bioscience and Human-Technology (NIBH), Agency of Industrial Science and Technology of the Ministry of International Trade and Industry (MITI).

For further information about Jun see under
`http://www.sra.co.jp/people/aoki/Jun/Main_e.htm`.

Metric	Value
Number Of Classes (NOC)	796[a]
Number Of Methods (NOM)	15'465
Number of attributes	2'003
Lines Of Code (LOC)	212433.03
Average LOC in a class	131.1315
Average NOM in a class	9.546296
Average LOC in a method	13.73637
Number of inheritance definitions	1'272
Number of accesses	151'522
Number of invocations	146'332

Table 4.3: Overview of the case study: Jun, version 398.

[a]Actually, only 635 classes belong to Jun. MOOSE must consider some auxiliary classes from The Smalltalk class library when generating a software model for consistency reasons.

4.5 Groups for Reverse Engineering

In this section we present a list of groups that we find valuable for the analysis of a software system under the perspective of design recovery, program understanding, and problem detection. The groups are visualised with *CodeCrawler*. The section is organized in two parts: class groups and method groups.

Before applying our groups to the underlying model we run a so called operator on the models entities. In MOOSE an operator iterates over all existing entities to compute and set values to attributes of the entity. Our operator is called `ClassBluePrintOperator` and maps the nodes' widths and heights to some metrics.

Remark that Moose and, especially in this section, CodeCrawler considers in its views the class instance side and the metaclass side as two separate classes. The metaclasses are labeled with the suffix `'_class'`.

Querying for example the system for the classes in the category `Jun-3dImage-Basic` would result into 2 classes, namely `Jun3dImage` and `Jun3dImage_class`, instead of only one: `Jun3dImage`. Presuppose therefore that if not written explicitly we always look at the class's instance side. To avoid problems with mixing up the two class sides we propose two approaches:

1. Filtering in advance all the classes belonging to the context the observer wants to look at. This works by defining a simple group. All following definitions will be based on the simple group.

2. Using class names the observer has to make sure he uses no wildcards at the name's end, like `Core.Object*`, because the group query can consider both classes in question.

In the following – as far as the class groups are concerned – we base our system investigation on "instance classes". We do that because usually the instance side of a class contains more methods and is therefore more representative for the class' behavior. For the definition of all class groups but the first we make use of the complement of the group *Metaclasses*[1]. This is the first group we present in this section.

[1]By complement we mean a group that contains all complementary classes.

4.5.1 Class Groups

Metaclasses

This is a simple group. With the complement of this group we provide the base for all the following class groups.

Values

components:	class nodes
selection criteria:	isMetaclass
edges:	*arbitrary*
display mode:	*arbitrary*

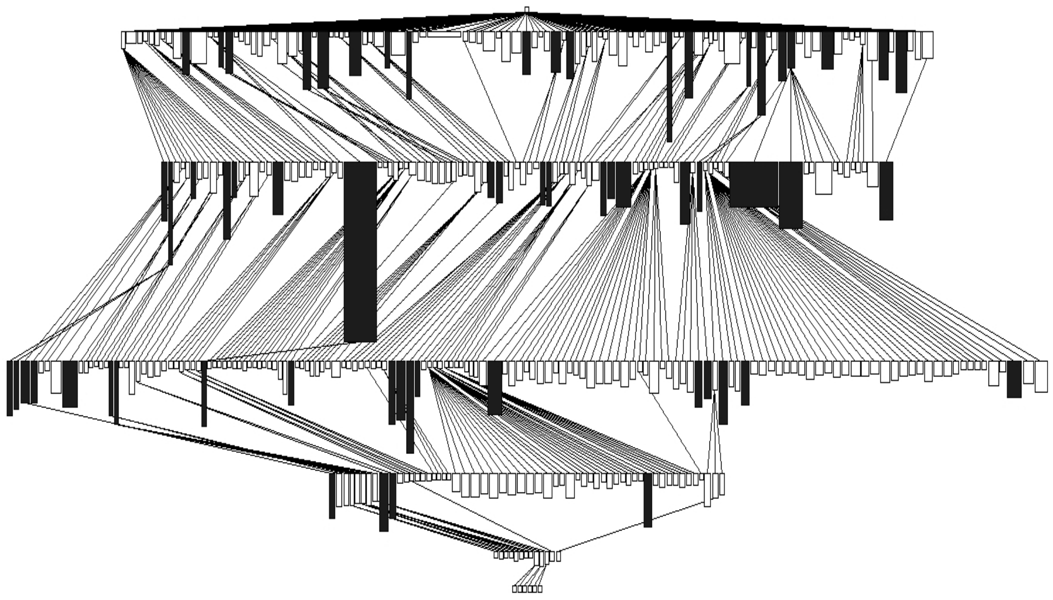
Intent. This group includes all meta-classes of the system. We want to use it for reducing the complexity of the subject system: either we remove its complement from the system to focus on the meta-classes or we drop the group itself to concentrate on the instance side of the classes.

Case Study

Resulting group size = 796.
We chose *color* for the display mode.

Figure 4.3 illustrates the hierarchy of the class `Core.Object`. We make a comparison of the two class sides.

Interpretation. In the two pictures of Figure 4.3 we colored all classes with the metric NOM greater than 30. Naturally, we see that the colored “instance classes” do not always correspond to the colored meta-classes. This is because a class’ number of methods on one side must not correspond to the other one. Now, picking the largest classes of the system would mean choosing those classes that are colored in both pictures.



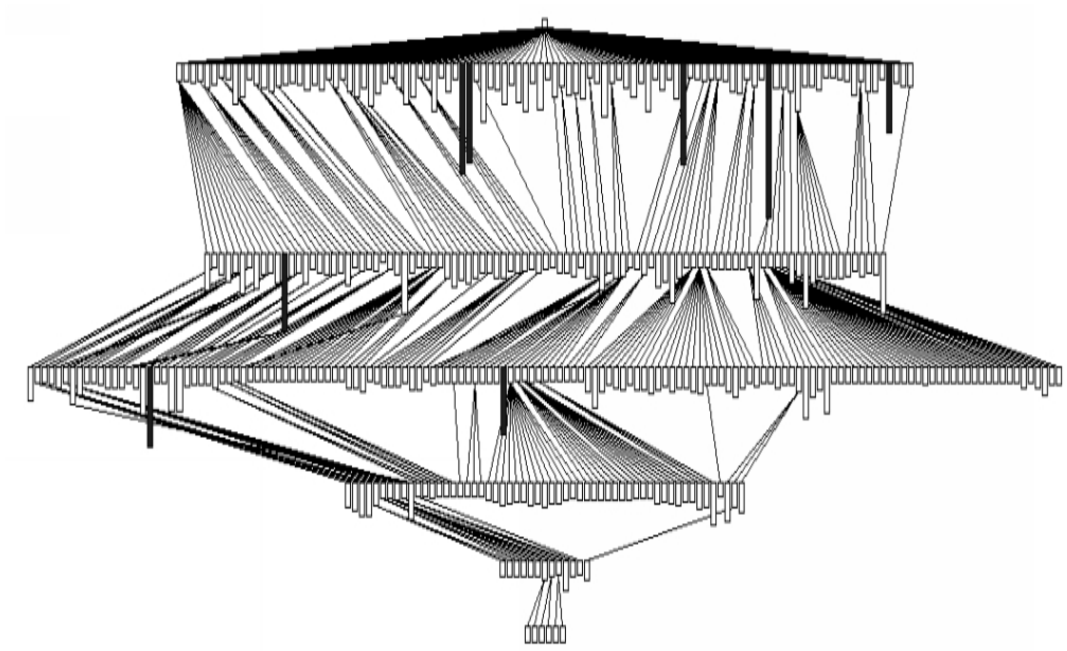


Figure 4.3: Above the metaclasses and below the instance side of the same classes. A class comparison with NOM > 30: the correspondent nodes are colored black.

Is Abstract

This is a composed group, based on the complement of the group *Metaclasses*.

Values

components:	class nodes
selection criteria:	isAbstract
edges:	inheritance edges
display mode:	color/selection

Intent. We would like to know how the abstract classes are distributed in the system, as represented in Figure 4.4. What are their subclasses? What do they look like, in terms of metrics?

All classes that respond positively to the query *isAbstract* are considered in the evaluation of this group. For representation a good visualisation mode is color or selection because it points out the position of the abstract classes in the class hierarchy. From the inheritance point of view it is rather uninteresting to group the classes in one node because we then lose their distribution.

Abstract classes are basically designed for subclassing. They provide a basic behavior for their subclasses. This is the reason why we want to keep the basic hierarchy structure.

Case Study

Resulting group size = 58.

For discussion we pick two classes that do not lie in the hierarchy of the class `Core.Object`, namely `JunOpenGLDisplayModel` and `JUNWinDIBSection`.

The abstract class `JunChartAbstract` is a subclass of the concrete class[2] named `JunOpenGLDisplayModel`. This constellation is represented in Figure 4.5.

In Figure 4.6 we see `JUNWinDIBSection`, an abstract class with 47 methods and only two subclasses counting just three methods each.

Interpretation. In the two cases presented above we have an implementation anomaly that should be considered in a later refactoring process.

[2]To be sure we checked in the source code if the class really is concrete.

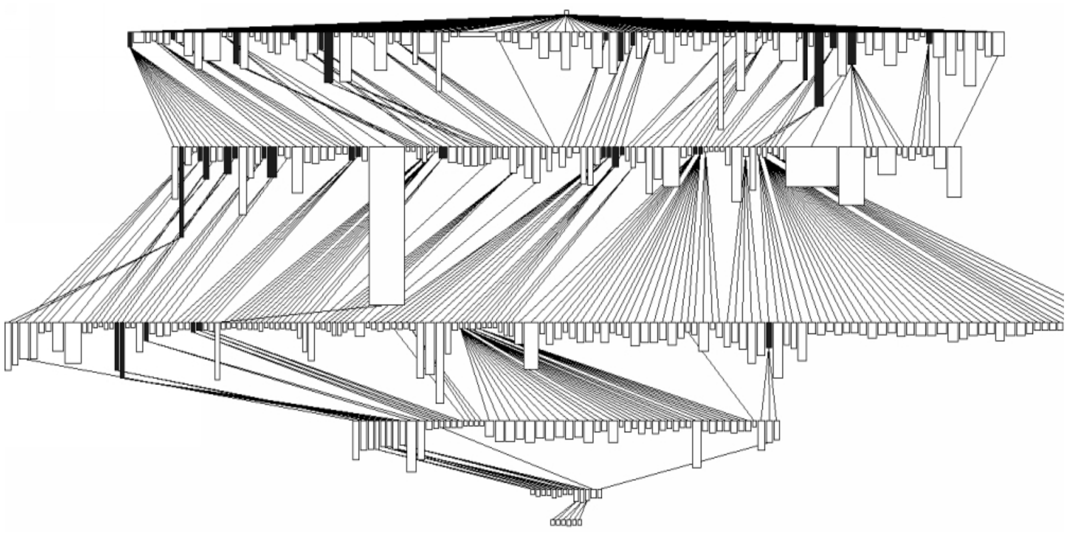


Figure 4.4: The items of the group *isAbstract* are colored black.

In the case of `JunChartAbstract` the question to ask is whether the constellation is reasonable. Basically, an abstract class generates no instances but its subclass – we call it a *concrete* class – can. In this very case the situation is opposite: we have a concrete class with an abstract subclass. This seems confusing. Why was `JunChartAbstract` chosen to be abstract? The developer should rethink the design and maybe make the superclass abstract, too, or aggregate both classes to one and only abstract class. The latter case is critical because the `JunOpenGLDisplayModel` has further subclasses that do not need the to implement the behavior of the class `JunChartAbstract`.

The situation of Figure 4.6 could lead to an aggregation of class `JUNWin-DIBSection` and its two subclasses, adding only six further methods to the big superclass. A further division of `JUNWinDIBSection` into more than two subclasses also represents a possible solution to reduce the complexity in the hierarchy.

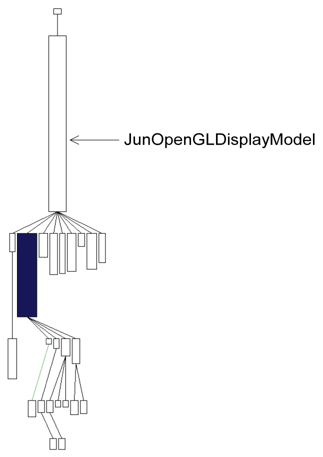


Figure 4.5: Abstract class JunChartAbstract is colored black.

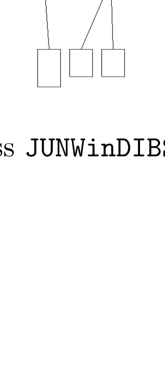


Figure 4.6: Abstract class `JUNWinDIBSection` is colored black.

Skips delegations

This is a composed group, based on the complement of the group *Metaclasses*.

Values

components:	class nodes
selection criteria:	hasMethodNamed: `m`
	¬ invokesMethodNamed: `m`
edges:	*arbitrary*
display mode:	color/selection

Intent. We wonder how a certain method is implemented. The group contains a subset of all classes that implement a method called `m`: those classes that do not send the same message (further) to another object. This happens when a class delegates (double dispatch) a message to another object as shown in Figure 4.7. Usually, the delegating class has no further implementation details regarding the method's function.

```
Jun3dParametricClosedCurve >> asJunOpenGL3dObject
     ^baseCurve asJunOpenGL3dObject
```

Figure 4.7: Jun3dParametricClosedCurve delegates the message to the attribute called `baseCurve`.

If a delegation takes place repetitively a sender chain gets created. Typically, the last link of the chain contains the concrete implementation. We show an example of a concrete implementor with Figure 4.8. Now, to skip those delegating classes we can apply our group. It includes only the classes that contain more code than delegating statements.

Case Study

For the variable `m` we choose the method name `asJunOpenGL3dObject`.
Resulting group size = 21.

Interpretation. This group contributes to the system understanding, actually it contains all classes that implement in a concrete way the method `m`. It skips all delegating classes that pass the message further to intermediate classes.

```
Jun3dParametricLineSegment >> asJunOpenGL3dObject
     ^JunOpenGL3dPolyline vertexes:
      (Array with:  self from with:  self to)
```

Figure 4.8: `Jun3dParametricLineSegment` is a concrete implementor of the method `asJunOpenGL3dObject`.

Interesting is the inspection of the hierarchy level where the classes with concrete methods are found. Normally, a subclass specializes some methods in the superclass. We conclude classes with concrete methods lie in the lower part of a hierarchy tree. Therefore, their superclasses are one or more levels above. We see in Figure 4.9 that most of the classes with concrete methods are on level three and higher[3].

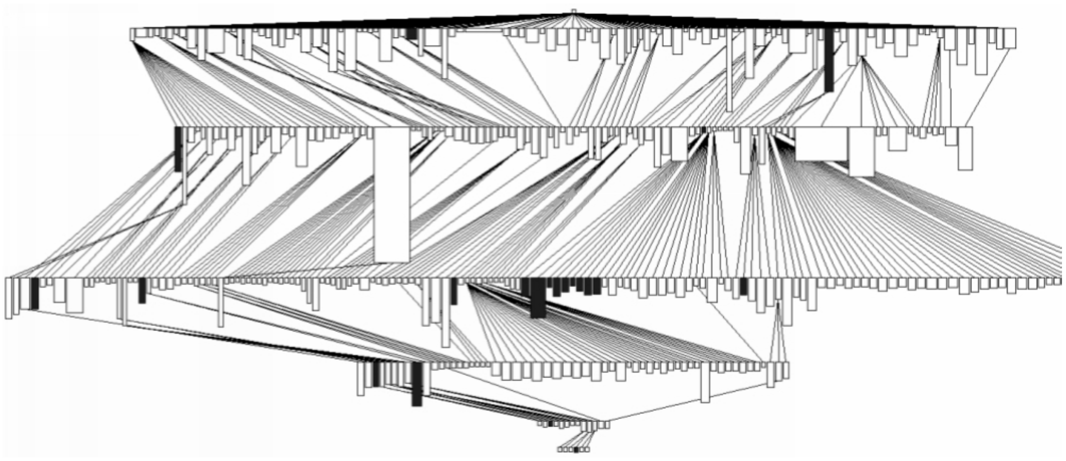


Figure 4.9: All classes that implement `asJunOpenGL3dObject` and do not delegate the message to another object are colored black.

[3]In a tree we count the levels beginning from the root node towards the leave nodes. A high level means a level in the lower part of a tree.

Implicit namespaces

This is a composed group, based on the complement of the group *Metaclasses*.

Values

components:	class nodes
selection criteria:	hasNameMatching: `aName`
edges:	inheritance edges
display mode:	aggregation

Intent. Although there are explicit namespaces in VisualWorks since version 7, programmers often introduce implicit namespaces using a special naming convention. They add to the class names unique prefixes. In most of the cases they are build by a few letters like `Lens`, `SUnit`, or `Jun`. Here an example of class names: `LensDataModel` instead of `DataModel`, `SUnitTest` instead of `Test`, or `JunPoint` instead of `Point`. Often the reason for implicit namespaces is that the class library of the development environment already contains classes with similar names. An example is the class `Point` in Smalltalk.

With this group we want to discover if the original developers have been consistent with naming and how namespaces coexist with other program structures like class categories. For instance we would like to compare a group A that represents a category with a group B that contains classes with a similar implicit namespace. Imagine for A the classes of the category named `Jun-Voronoi-2d-Diagram` and for B all classes with the implicit namespace `JunVoronoi`. Do the two groups contain the same classes?

Case Study

As `aName` we chose the following category prefixes:
Jun-3dImage*
Jun-Breps*
Jun-Collections*
Jun-Csg*
Jun-Delaunay*
Jun-Dxf*
Jun-Geometry*
Jun-Goodies*
Jun-Metaball*
Jun-Octree*

Jun-OpenGL*
Jun-Parametric*
Jun-Solid*
Jun-System*
Jun-Terrain*
Jun-Topology*
Jun-Voronoi*
Jun-Vrml*
Jun-Win32*.

The application results in 16 groups with the number of classes between 3 and 74.

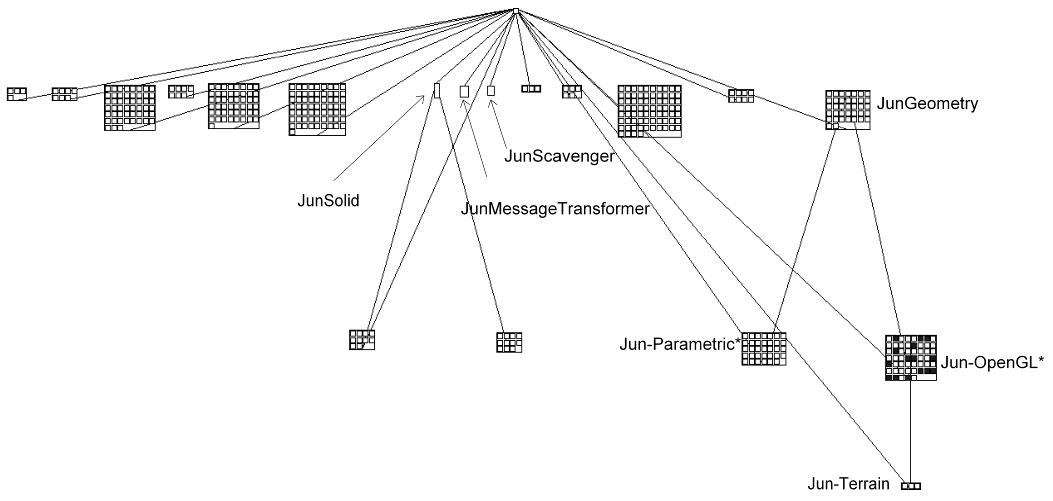


Figure 4.10: All categories within the hierarchy tree of `Core.Object`. For `Jun--Open-GL*` the black colored nodes show where no naming convention was applied.

In the resulting view three classes remain isolated: `JunMessageTransformer` from the category `Jun-Kernel-Method`, `JunSolid` contained in the category `Jun-Solid-Abstract`, and `JunScavenger` in `Jun-System-Support`. The first two are interesting because of their categories' size: both categories contain only one class, those mentioned above. Class `JunScavenger` was not considered for a group because the other classes in the same category depend on a different hierarchy.

The edges between the class node and the group nodes point out the inheri-

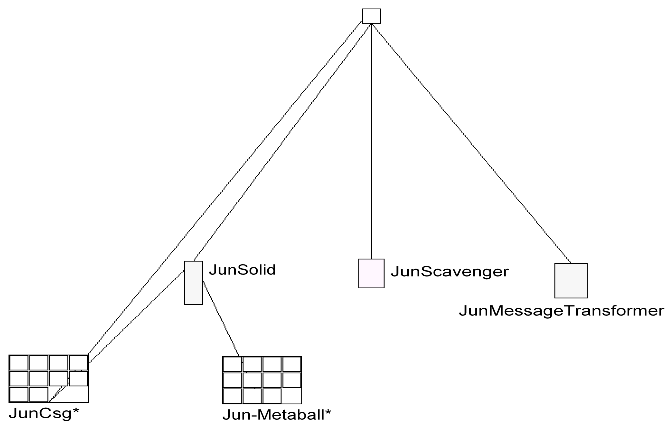


Figure 4.11: Except of `JunScavenger` the class nodes are classes that belong to categories with no more other classes.

tance relationships. The class node `JunSolid` is connected to the namespace-groups `Jun-Csg*` and `Jun-Metaball*` as we can see in Figure 4.11.

Another particularity that gets its attention is the relationship between the groups `Jun-Geometry*` and `Jun-OpenGL*`. Since we used inheritance edges in the definition of the groups we know that the `Jun-OpenGL*` inherits from `Jun-Geometry*`. So we have hidden dependencies between the two namespaces.

Interpretation. The situation of the case study should help the developers to rethink the "existence" of the categories `Jun-Kernel-Method` and `Jun-Solid-Abstract` because of their little size. This would reduce the complexity of the category structure in the system. An alternative refactoring could be moving, where appropriate, the classes to other categories.

By inspecting the inheritance relationship in Figure 4.10 between the groups `Jun-Geometry*` and `Jun-OpenGL*` we find one inheritance definition. It represents the connection between the class `JunCoordinateAngles` from the category named `Jun-OpenGL-Flux` and the class `JunGeometry` from category `Jun-Geometry-Abstract`. By the way `Jun-OpenGL-Flux` contains classes with names beginning with `JunOpenGLFlux` except `JunCoordinateAngles`. Here we have an example where the implicit namespace was not applied to the class' name. Moving the latter class into another category, *e.g.*, one of those named `Jun-Geometry*`, is a possible restructuring.

For Figure 4.10 we colored the classes that have an implicit namespace that does not correspond to the category. This situation expresses in what relations implicit namespaces coexist with the categories. We see in the picture a lot of blue colored nodes. We believe that all nodes should be white because then we would have a consistent system structure.

Subclasses

This is a composed group, based on the complement of the group *Metaclasses*.

Values

components:	class nodes
selection criteria:	inheritsInHierarchyFromClassNamed: `aName`
	¬ hasNameMatching: `aName` [a]
edges:	inheritance edges
display mode:	aggregation

[a] If we use only *inheritsInHierarchyFromClassNamed:* `aName` as selection criterion the result also includes the class called `aName`.

Intent. This definition is useful to reduce the system's complexity and for the definition of composed groups.

The purpose of this group is to inspect common properties of classes organized in the same hierarchy tree with a given root (`aName`). We want to focus on properties that express the groups' sizes in terms of `NOM` or `LOC`.

Case Study

As `aName` we choose all subclasses of the class `Core.Object` at the hierarchy nesting level (HNL) = 1. `Core.Object` counts 104 direct subclasses.
The resulting group sizes vary between 1 and 65 items.

In Figure 4.12 we see a number of classes that was not affected by the grouping. This is because they have no subclasses, *e.g.*, in the view the class nodes have no subtree. The core class `Object` contains 70 direct subclasses that have no subclasses. Because of that our group definition above creates only 34 groups. The remaining 70 classes are not considered for a group creation[4].

Interpretation. The group is useful for the program understanding. It helps to localize subtrees in complex class hierarchies. Moreover, the display mode *aggregate/collapsing* reduces considerably the system's complexity.

The largest classes not affected by the grouping are `JunSolidModelingEngine` with NOM = 63 and `JunEncyclopedia` with NOM = 107. Both classes are not abstract. Since they do not have any subclasses we should think of the reason why the classes were implemented with this number of methods. A look at their names reveals a complex functionality and of course this justifies

[4] We remember that the grouping of one element is the element itself (see Section 3.2.2).

a large class structure. Anyway, a reengineer might consider to split one of these classes for a refactoring if there is an appropriate design behind.

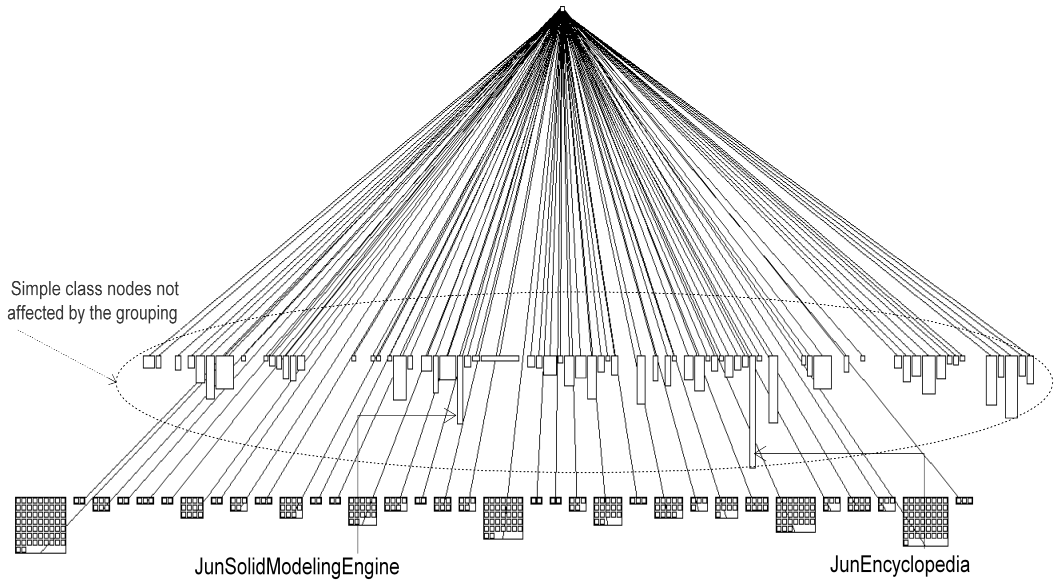


Figure 4.12: All classes in the `Object` hierarchy grouped to direct subclasses. The nodes are shifted for a clearer view.

Intentional layers

This is a composed group, based on the complement of the group *Metaclasses*. We want to investigate the instance side of system classes.

Values

components:	class nodes
selection criteria:	hierarchyNestingLevel = `hnl`
	hasSourceAnchorMatchingName: `aName`
edges:	inheritance edges
display mode:	aggregation / collapsing

Intent. The idea is to analyse tree levels of a hierarchy tree from a particular point of view, the class category. Practically, we project the category structure on the tree. For each tree level in a class hierarchy we group all classes belonging to the same category. We then inspect the relationships between the groups. Above all we want to detect dependencies between the categories. We would like to inspect the dependencies in order to to find out the reason of their existence. We also believe that with the result one can recognize modules or extract information on arbitrary layering in the class architecture.

Case Study

We set for `hnl` the values from 1 to 5, each corresponding to a tree level of the `Core.Object` hierarchy tree.
For every value of `hnl` we choose as `aName` the following category names:
Jun-3dImage*
Jun-Breps*
Jun-Collections*
Jun-Csg*
Jun-Delaunay*
Jun-Dxf*
Jun-Geometry*
Jun-Goodies*
Jun-Metaball*
Jun-Octree*
Jun-OpenGL*
Jun-Parametric*
Jun-Solid*

Jun-System*
Jun-Terrain*
Jun-Topology*
Jun-Voronoi*
Jun-Vrml*
Jun-Win32*.

Resulting groups = 41. Their sizes vary between 1 and 47 items.

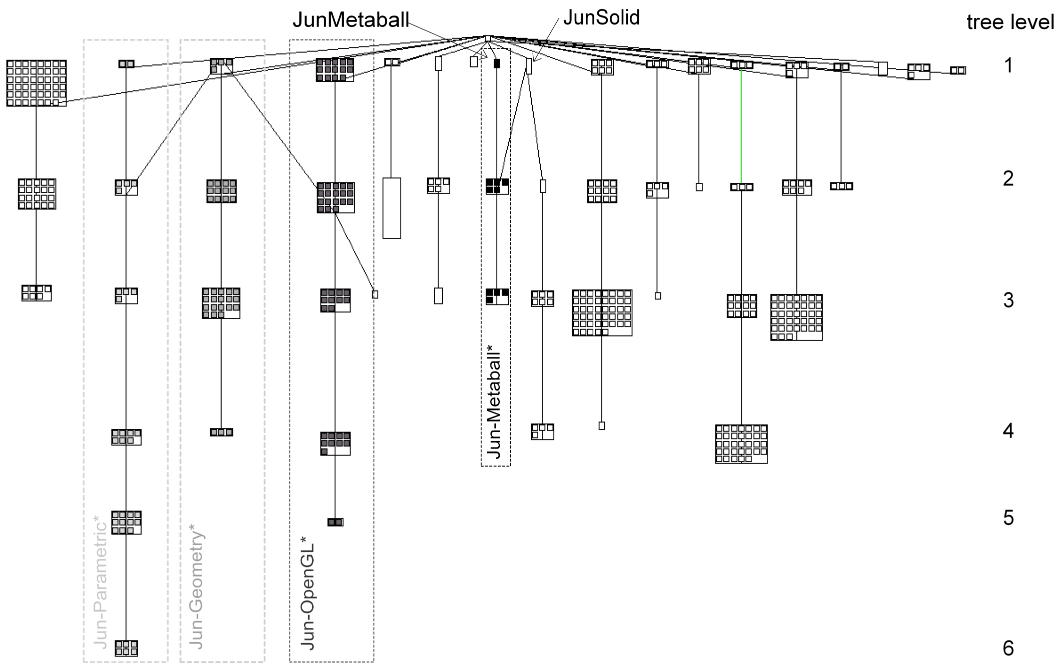


Figure 4.13: Object hierarchy structured in class categories.

The result in Figure 4.13 shows us a well ordered tree except for some nodes. The group node representing the categories `Jun-OpenGL*` at the tree level two, for example, is connected with two nodes at level one. Namely the group `Jun-OpenGL*` and `Jun-Geometry*`, both at the level one. By inspecting the edges we find out that the edge representing the relationship with `Jun-Geometry*` stands for the inheritanceDefinition between the class `Jun-Geometry` and its subclass `JunCoordinateAngles`. The other edge represents inheritance definitions within the same category. Therefore we do not investigate them in detail.

A similar situation exists with the group `Jun-Metaball*`, at tree level two. It has connection edges with the class `JunSolid` and the class named `JunMetaball`, both at the first tree level. This is because the class `JunMetaballSolid`, contained in the categories `Jun-Metaball*`, is a subclass of `JunSolid` and all others have the superclass `JunMetaball`. We draw attention to the existence of the category `Jun-Metaball-Solid`.

The last situation involves the category named `Jun-Parametric*`, at level two. Two groups are connected with it, namely `Jun-Parametric*` and `Jun--Geometry*`. Again we neglect the relationship in the same category. We concentrate on the edge between `Jun-Geometry*` and `Jun-Parametric*`. The edge contains two inheritance edges: `JunGeometry` is superclass of the classes named `JunParametricGeometry` and `JunPointOnEntity`. Both subclasses are contained in the group `Jun-Parametric*`.

Interpretation. We focus our analysis on nodes that are connected with more than on node on an upper level as demonstrated in Figure 4.14. We call these connections *unary* respectively *n-ary*.

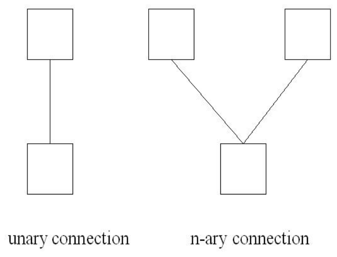


Figure 4.14: Example of node connections.

We found n-ary connections with the groups `Jun-Parametric*`, `Jun-OpenGL*`, and `Jun-Metaball*`. They all contain classes with superclasses in other categories. We think that the investigation of such dependencies can reveal some anomalous dependencies that do not reflect of the original design. The results can support the decision whether to move the classes in question to other categories or if a deletion makes sense?.

Little behavior addition

This is a composed group, based on the complement of the group *Metaclasses*[5].

Values

components:	class nodes
selection criteria:	inheritsHierarchyFromClassMatchingName: `aName`, NMA < `nma`, NME < `nme`
	hasSubclasses: false
edges:	*arbitrary*
display mode:	color / selection

Intent. In order to obtain a more expressive result we choose for `aName` an abstract class. Therefore, we expect them to have subclasses since abstract classes typically provide a basic behavior for their subclasses. Normally, abstract classes are supposed to be extended with their subclasses. This can be done by specializing their methods and with the addition of new methods.

We would like to know what the typical number of subclasses is, in first place. On the other hand, we are interested in measuring the metrics of the group created with an abstract class and its subclasses. For this purpose it makes sense to aggregate or collapse the focused items.

Case Study

For `aName` we chose the names of all abstract classes in the system, 58 classes in all.
We set `nma` to 2 and `nme` to 2.
The values created 12 groups with sizes between 2 and 20.

Interpretation. With this kind of group we detect abnormal design. In particular we find classes with just few changes comparing to their abstract superclasses. The observer should ask himself when and if the subclassing makes sense. As alternative we propose to aggregate the inspected classes, where reasonable.

We picked from the result the class `JunAbstractOperator`. It has six subclasses that only add few changes to the behavior of the superclass. In Figure 4.15 we see that implementation and collocation of these classes is reasonable because of the further subclassing.

[5]In this case it is also based on the group called *Is Abstract* that we introduced previously.

If we focus on `JunEulerianAngles` the constellation is different because we find only two subclasses with no classes at deeper hierarchy level. The groups' NOM is equal to 5 and this means that an aggregation with the superclass would create a class with just five methods more than `JunEulerianAngles` has. Of course, the new class should then be implemented as a concrete class because it would not have any subclasses. Considering the large size of `JunEulerianAngles`, namely 26 methods, we do not approve a class aggregation respectively a class enlargement.

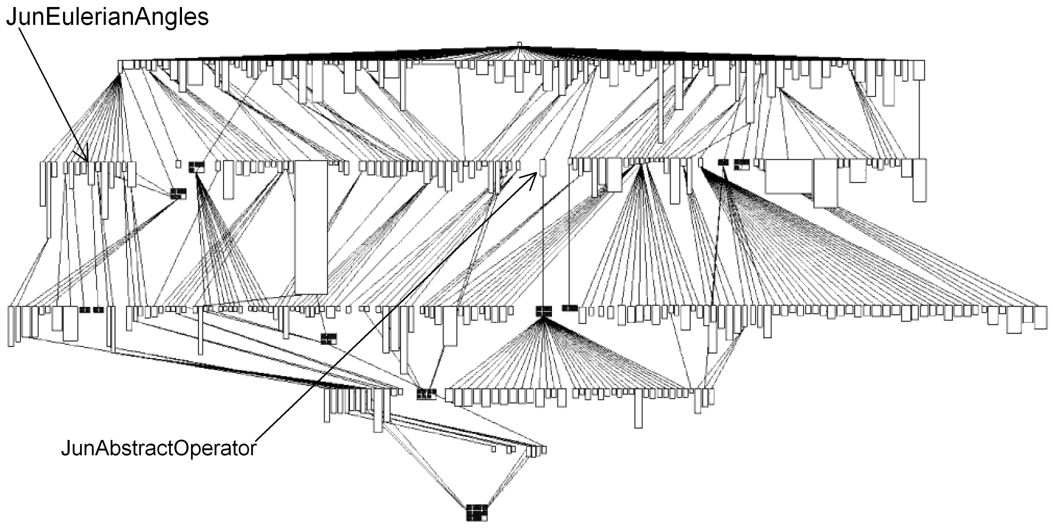


Figure 4.15: The class with little behavior changes are grouped and colored.

4.5.2 Method Groups

Large methods

This is a composed group. To see a relation between large methods and their classes we group – in a preprocessing step – all methods belonging to the same class. As result we see group nodes representing classes displaying the contained method nodes. Figure 4.16 illustrates an overview of all methods grouped as classes. In the upper left corner we see a number of simple nodes. They were not grouped because they belong to classes with a single method.

Values

components:	method nodes
selection criteria:	belongsToClassWithNameMatching: `aName` LOC > `loc`, MSG > `msg`, NI > `ni`, NOS > `nos`
edges:	*arbitrary*
display mode:	color /selection

Intent. The resulting view displays large methods based on the values itemized above. Of course by varying the limiting values the size of the created group can be dramatically changed. A good approach for obtaining large methods is to change the limiting value according to a function that uses the average value of the metric *Lines Of Code* (LOC) of all the methods in the system.

We want to discover in what classes large methods lie and see a comparison between the *Number Of Methods* (NOM) of the same class. By coloring the group – all the large methods – we also wonder if there are fully colored nodes. This would be an indication of a large class if the number of methods is high, too.

Case Study

For `aName` we took the names of all classes in the system. Further we set:
`loc` = 20
`msg` = 20
`ni` = 20
`nos` = 20
Resulting group size = 756.

At first sight, the group size seems very large. If we consider the system size based on the NOM – that is 15'465 – the size of the group is not so surprising. So we can say that 4,9% of all methods in the system are large.

Since the system is large, 1620 classes, and the resulting view is difficult to interpret we propose in Figure 4.17 a zoomed cutout of the entire system. The original view was a scatterplot[6]. based on the nodes in Figure 4.16. We picked the classes named `JunBrepsHalfEdge` and `JunMoviePlayer` to discuss their properties. The first has NOM equal 58 and the latter NOM equal 65.

We see that the classes with large methods have also a large number of methods (NOM).

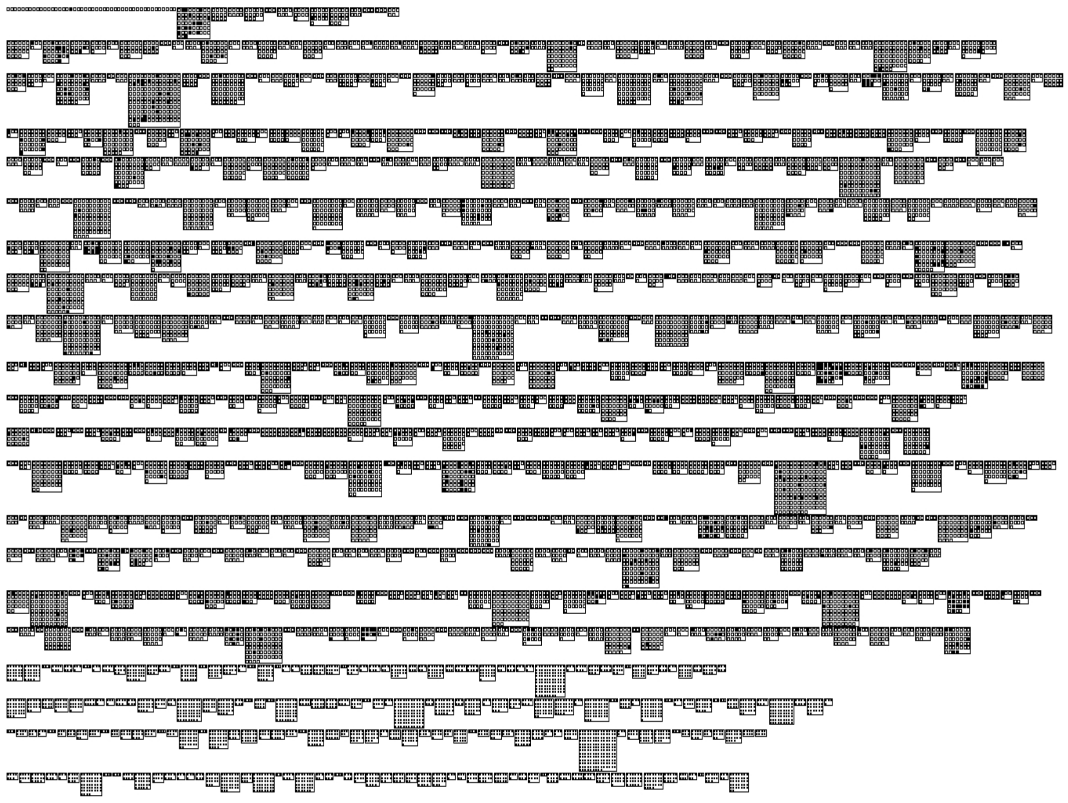


Figure 4.16: Overview of all classes displayed with their methods. The system is too big to see the black colored large methods.

Interpretation. We detect design anomalies. Generally, classes should not contain large methods. This adds more complexity to the class. Additionally, if the class has also a high number of methods we should consider a splitting of the

[6]The plot uses LOC and NOS as position metrics for the class nodes.

class for refactoring.

As far as classes with one method are concerned programmers should think of a refactoring, too. At least it is interesting to know why such little classes have been implemented? Are they direct subclasses of `Object` or can they be aggregated with their superclasses?

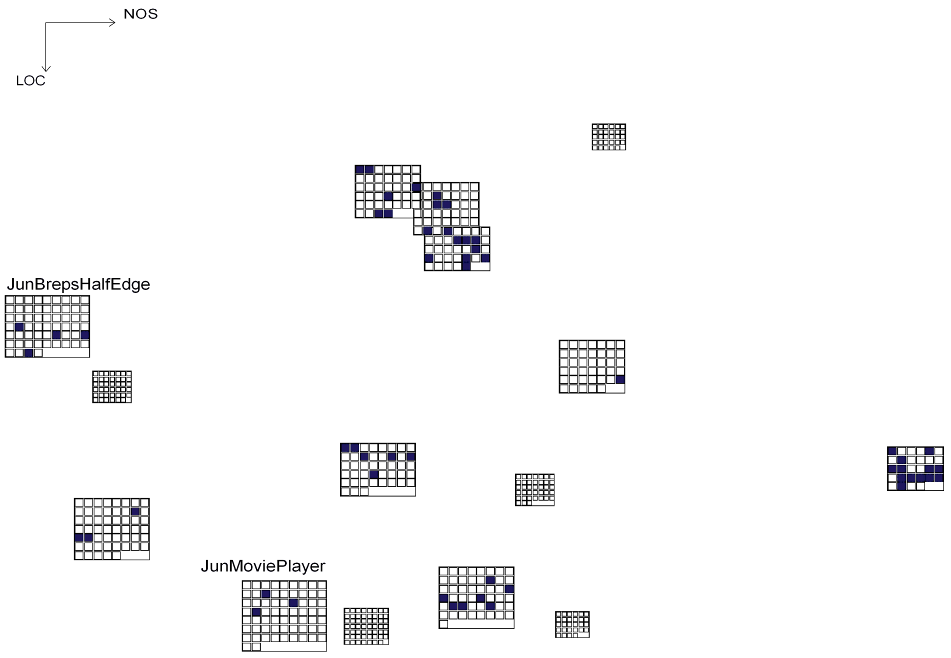


Figure 4.17: Classes with large methods have also a high number of methods. This is a cutout of a scatterplot. To reduce the system complexity the nodes in Figure 4.16 were aligned using NOS and LOC as position metrics. Some nodes overlap because the expansion of the scatterplot depends on the system's outliers and not on the number of displayed nodes.

Category Interface

This is a simple group definition.

Values

components:	method nodes
selection criteria:	a) belongsToClassWithSourceAnchorWithNameMatching: `a`
	b) belongsToClassWithSourceAnchorWithNameMatching: `b`
edges:	invocation edges
display mode:	aggregation node

Intent. The obtained view visualizes the invocations between two groups of methods belonging to different namespaces. As namespaces we imagine any kind of strings that describe abstract components, packages, class categories, or class names[7]. In this case we use names of class categories. The set of invocations between the resulting groups describe the interface between two categories. More precisely, the method nodes at the end of the edges form the interface for the opposite group. Figure 4.18 illustrates an example.

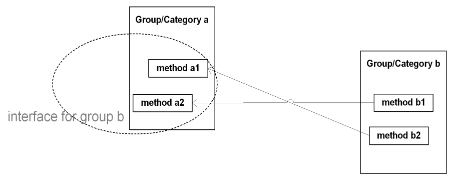


Figure 4.18: The method nodes at the end of the invocation edges form the interface for the opposite group.

Since the number of methods in the groups is often relatively high and the number of invocation edges even higher, the connections between the method nodes are distracting. A solution to handle the complexity of the current view is to represent the group visually with a node. That reduces the complexity of the edges since the edges within the node are hidden. Furthermore, the incoming and the outgoing edges of a group node get collected in composite edges. In Figure 4.20 we collapsed all the classes belonging to a category.

[7]Later we present with the next group called *Class Interface*.

The resulting node hides the invocations within the category. Therefore the analysis of the invocation edges between both categories is better facilitated.

Case Study

For `a` and `b` we set the names of the following class categories:
Jun-Breps-Abstract
Jun-Breps-EulerOperators
Jun-Breps-GeometricOperators
Jun-Breps-Objects
Jun-Breps-Operators-Abstract
Jun-Breps-Support
Jun-Breps-UtilityOperators.

In the case `a` = `Jun-Delaunay-2d` we obtain a group of 206 methods. For `b` = `Jun-Voronoi-2d-Diagramm` the resulting groups has got 220 methods. The categories are represented in Figure 4.19.

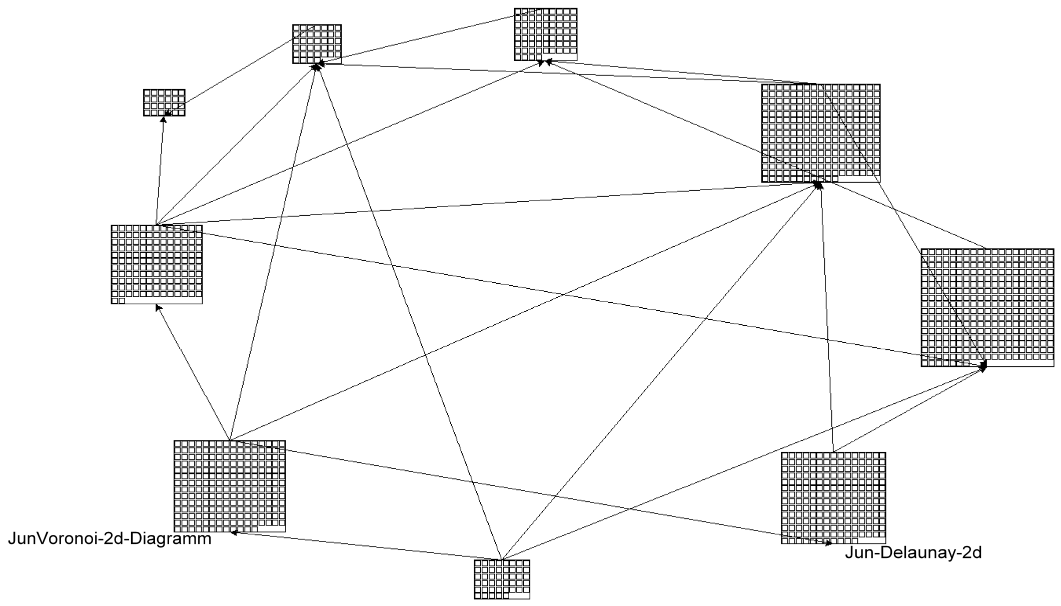


Figure 4.19: Methods are grouped according to the category of their class. The edges between the groups represent the method invocations.

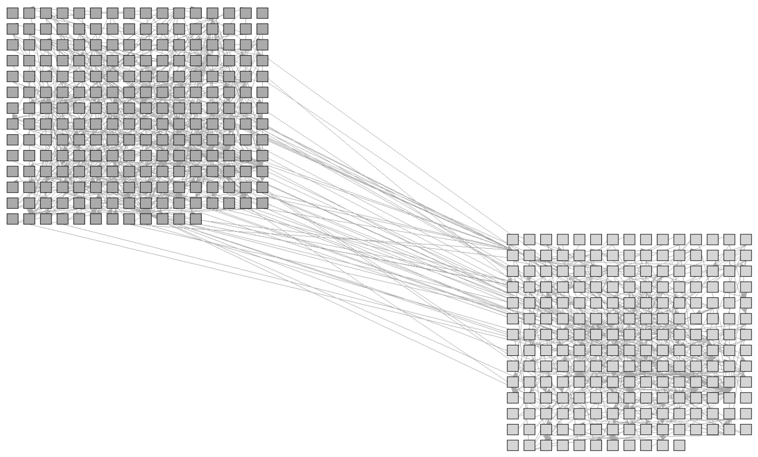


Figure 4.20: A colored category interface. The light gray method nodes belong to the category `Jun-Delaunay-2d`, the gray nodes correspond to methods in the category `Jun-Voronoi-2d-Diagram`.

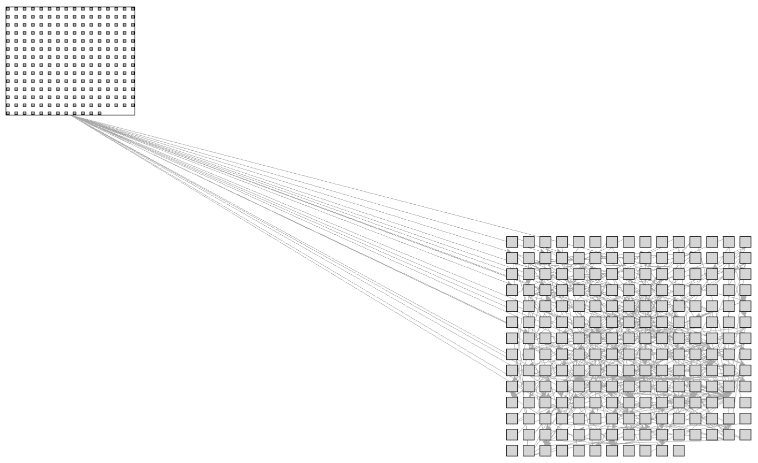


Figure 4.21: The method nodes of category `Jun-Voronoi-2d-Diagram` are grouped. The edges starting from the group node give us information about the interface of `Jun-Delaunay-2d`.

Interpretation. With this group definition we can detect abnormal dependencies, *e.g.*, when connections exist that the design does not prescribe. Furthermore, the identification of an interface is useful for program understanding.

Class Interface

This group represents a modification of the previous group called *Category interface*. The group nodes can be seen as classes.

Values

components:	method nodes
selection criteria:	a) belongsToClassNamed: `aNameA`
	b) belongsToClassNamed: `aNameB`
edges:	invocation edges
display mode:	aggregation node

Intent. The purpose of this group is to detect the communication between two classes and the identification of their interface.

Case Study

As starting view we pick the result of the view returned from previous group, *Category Interface*. We decided to inspect the same situation at class level.

For discussion we take a closer look at the classes in the categories `Jun-Delaunay-2d` and `Jun-Voronoi-2d-Diagramm`.

Resulting class groups = 17.

Interpretation. Again we can detect equivocal invocations. For instance, in Figure 4.22 we see the method node `isBoundaryVertex` that has no invocation edge that connects it to any other node.

More conspicuous is the same situation with a whole class, namely `JunVoronoi2dTriangle`. None of its methods has either outgoing nor ingoing invocation edges. Of course, in Smalltalk it can happen that senders of a methods are not detectable because message sending can also be performed with symbols. But this situation states that any method in the class neither invokes a method nor is invoked by another one. This is a point that needs more investigation.

Apparently, there are also classes that act as ”invoker”. For example, the class `Jun2dDelaunayList` seems to have only methods that invoke other methods. We find also the inverse situation: `JunVoronoi2dProcessor` has only methods that are invoked.

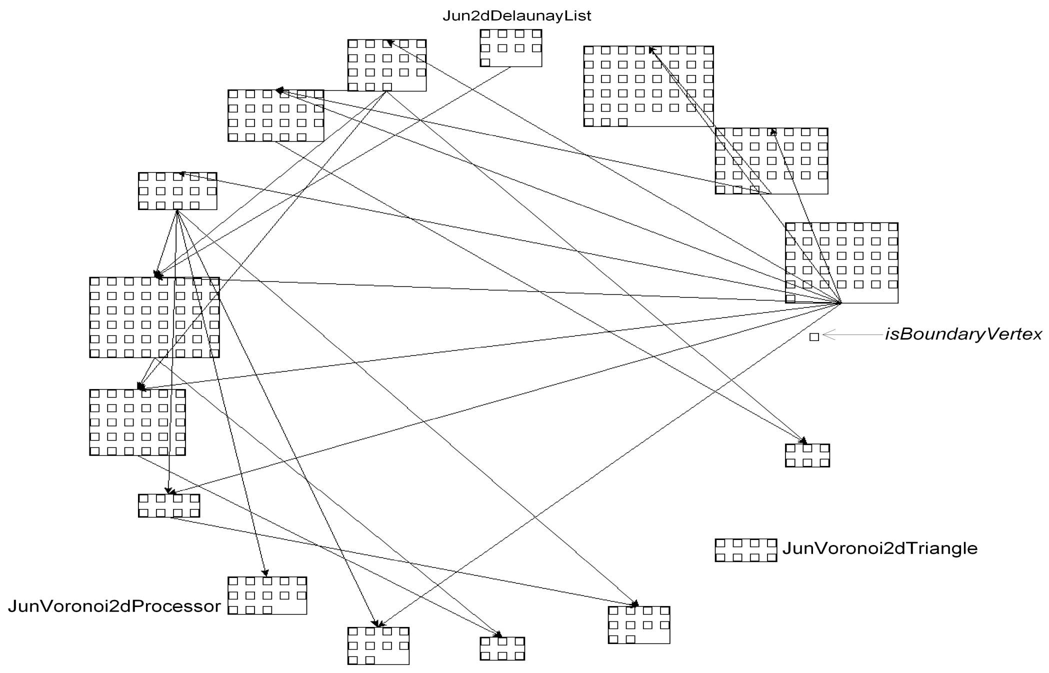


Figure 4.22: Invocations between classes from the categories `Jun-Voronoi-2d-Diagramm` and `Jun-Delaunay-2d`.

Category methods

Values

components:	method nodes
selection criteria:	sourceAnchor = `aName`
edges:	invocation edges
display mode:	*arbitrary*

Intent. This is a composed group. The subdivision of methods into categories[8] is known in Smalltalk. We want to visualize this classification.

Case Study

We set aName to `functions`.
Resulting group size = 348.

We defined the group on all methods belonging to classes in the category `Jun-Delaunay-2d`. Figure 4.23 shows all the methods in the class category and the blue colored "function methods". The pictures illustrates two things clearly: 1) the proportion of these methods and 2) what other methods they invoke.

We can also see some standalone nodes without any connections. There are a few reasons for that: either the correspondent invoked respectively the invoking methods are not in the current view or they are not used.

Interpretation. In Figure 4.24 we picked all methods in the class category `Jun--Delaunay-2d` and grouped them to classes. There we can make statements about two classes: `Jun2dDelaunayList` and `Jun2dDelaunayElement`. Neither of them have a method category called `functions`.

[8]Note: Here, we refer to method categories. We do not speak of the same categories meant in the section 4.5.1.

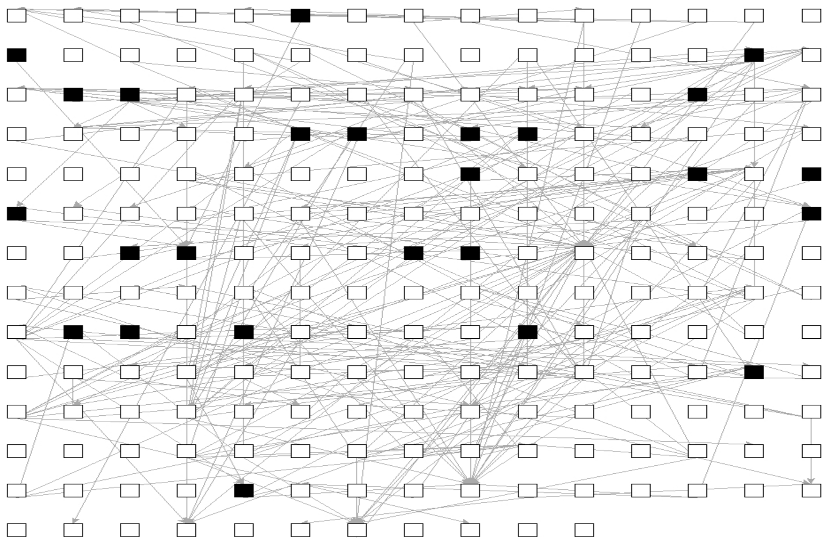


Figure 4.23: All methods in the class category `Jun-Delaunay-2d`. The black colored method nodes belong to the method category called `functions`.

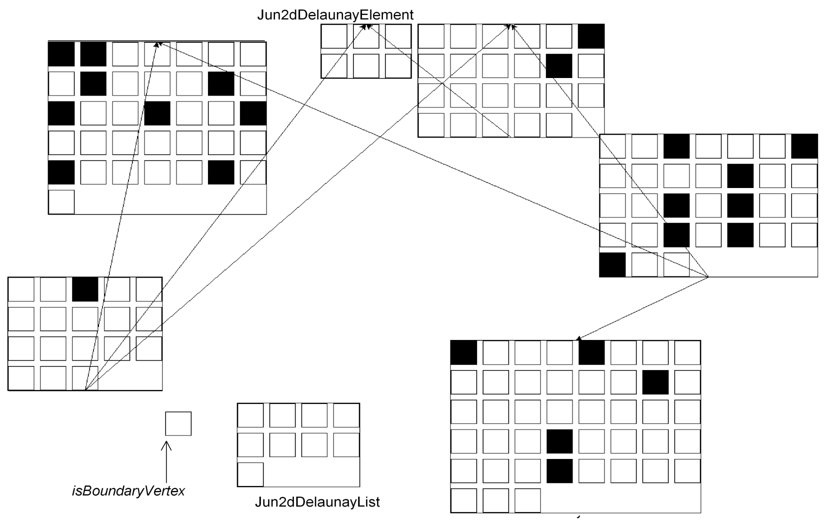


Figure 4.24: The method nodes of Figure 4.23 are grouped to classes.

Chapter 5

Conclusion

5.1 Summary

This master's thesis proposes an object-oriented reverse engineering approach based on grouping. The intention of grouping is the creation of groups with components of a software system. In particular, in this work the system is represented by a software model that conforms to the FAMIX model. Our approach follows two paths: we describe the logical grouping and we discuss the graphical representation of groups. Grouping allows us to create different abstraction levels and alternate views of a software system.

Our discussion starts with automated support in the reverse engineering since it alleviates the managing of complex and large software systems. Many reverse engineering tools create graphical representations of software systems to support the program understanding and the design recovery. Some of them allow one filtering and editing graphs to identify subsystems. For instance, the filtering of *Rigi* [Mue86] is based on node- and arc types – the latter represent relationships – and on user selection. The selected item can be collapsed and it may again be expanded. The selection of items by indicating some properties works only with attributes, structures and names.

The basic idea of our approach is to offer a simple working grouping facility that enables definition, generation, representation, and management of user defined as well as predefined groups. The groups' definition is based on the underlying model's components. The purpose is to give the user a tool to access the subject system and cluster its entities arbitrarily.

This work formalizes the logical creation of abstract groups. The grouping is split into two parts: the query-based component selection and the creation of the group. A query-based selection requires detailed information about the components in the software model. Once a group is created this information has to be propagated to the group in order that it can be examined as abstract system entity. A group has to provide the same attributes that its items share. Moreover, the underlying model has to provide an effective query interface that enables the access to all attribute values of its components.

We discuss the problems that are bound to our approach. After the discussion of the prerequisites to implement our approach by using *CodeCrawler* [DDL99, Lan03] – a reverse engineering tool that supports program visualisation – we present our implemented extension called *Classifier* (for details see A.1).

With Classifier we introduce interactivity to make the groups definition flexible and adaptable to user related needs. From this preparations we are able to extract some groups effective in the support of reverse engineering software systems. For the analysis of the effectiveness of our grouping approach we discuss these groups and their results after application on a concrete software system.

The main contributions of our work are:

- Simplicity and Scalability. We give an informal definition of logical grouping and graphical representation of groups without making use of complex algorithms or composed metrics. The query-based logical grouping is effective and is valuable for using the static information of a software model.

  Our approach is scalable and can be applied on all kinds of system components. This involves an enormous reduction of complexity valuable to reverse engineer even very large systems.

- A list of groups to support the reverse engineering of software systems. With there application we are able to obtain several useful results, which address either program understanding or problem detection.

- Interactivity. We implement an application called *Classifier*. It provides all the grouping procedures and combines two graphical user interfaces for the creation, the management, and the visualisation of groups.

5.2 Improvements

On some occasion different ideas of improvements for future work arised. We tried to order them to give some advice.

5.2.1 Classifier

Ability to store steps is needed. ... Useful is the possibility of repeating the process automatically and interactively analysing the dependencies of the architectural components. [Riv00]

- Extend the grouping algorithm. In particular add automated group generation, *e.g.*, `"for all packages group all classes that belong to their package"`. Right now the user has to note all the package names and then create the groups manually using the names. By automating such procedures a lot of time can be saved in the group generation.
- Nesting groups, *e.g.*, the creation of composite groups may be affected by previously generated groups. Simple groups often become a definition base for future groups.
- Of course, extending the list of groups that offers a basis for reverse engineering a software system. Predefined as well as newly defined groups can always be modified or extended.

5.2.2 Moose

- Introduction of group types like *homogeneous* - class group, method group, attribute group - and *heterogeneous*. For the homogeneous types, groups can be treated as fictive classes, methods and so forth. The types implement a more effective query interface using all the information connected to its type that corresponds to its components type.
- Add more entities, *e.g.*, package, category, file *etc.*. Of course this enlarges the model adding more complexity.
- Add more software metrics to give the entities more information. This measure enlarges the scope for the group definition since it is based on the attributes of the model's components. Furthermore, a larger scope reduces the complexity of the results.
- Add types to the model[1]. Introduce type inferring algorithms that determine an object's type that can be mapped to the object's attributes. There are some type determining programs in Smalltalk like Vassily Bykov's *type collector* but

[1]This a special case for Smalltalk or more generally when treating untyped programming languages. In all other cases like C++ and Java this point can be neglected.

this can only determine types when the objects are actually present in memory. We are working on a type inferring system that works on the static FAMIX model. It will be soon published in the master's thesis of Tobias Aebi.

5.3 Upshot

The software re-engineering of *legacy systems* represents today a major concern in software industry. This work is about grouping in reverse engineering. Companies have to schedule time and money to provide personnel that attempts to create an abstract representation of the systems' high-level functionality. We think that our approach of grouping and the discussion of the groups presented in chapter 4 can be useful applied in the reengineering process, an issue in software industry.

Appendix A

Realisation

In this chapter we present the implementation of *Classifier* and the interface with the underlying architecture. It is based on *CodeCrawler*, *i.e.*, a reverse engineering tool that provides a number of functionalities like program visualisation and graph manipulation. Like CodeCrawler Classifier runs on VisualWorks Smalltalk version 7.

A.1 Classifier

Classifier is implemented as an extension of CodeCrawler. Figure A.1 illustrates the component structure of the system. The most important classes of Classifier are listed in section A.5.

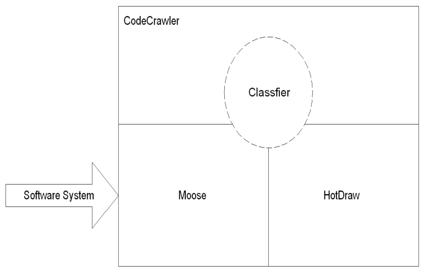


Figure A.1: Moose, CodeCrawler, and Classifier.

For the user two graphical user interfaces are important, namely the *GroupGenerator* and the *GroupManager*. The group generator is illustrated by Figure A.4; it provides basically the functionality for defining groups. In Figure A.5 we see the GroupManager that is created to visualize the groups within the views displayed in the CodeCrawler windows.

In the following sections we show the connections points of Classifier with the remaining architecture.

A.2 Connection to Moose

> *FAMIX does not support grouping explicitly, i.e., there is no concept in FAMIX to capture and exchange user-defined groups of model elements. The reference schema underlying FAMIX, however, supports grouping by providing unique identification for any element in a model.* [Tic01]

MOOSE is the Smalltalk implementation of the FAMIX meta-model. We see the core hierarchy in Figure A.2.

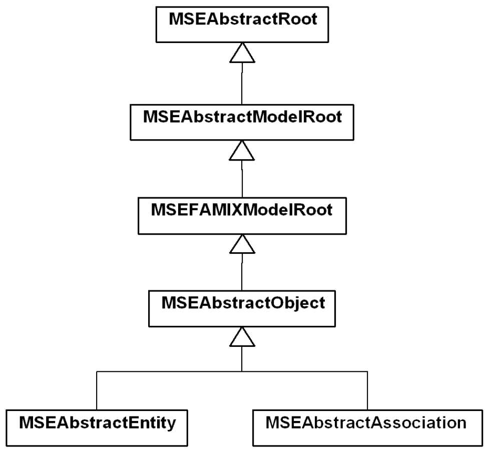


Figure A.2: MOOSE. Entities and associations of the FAMIX model are implemented with the classes `MSEAbstractEntity` and `MSEAbstractAssociation`.

The model serves primarily as knowledge base for Classifier. It returns the components Classifier wants to represent, enables the calculation of metrics, and responds to queries. The connection between Classifier and Moose is given via *CCNode* and `CCEdge`, two CodeCrawler classes. They represent entities respectively associations of the meta-model with a direct reference. We see the structure of entities and associations in Figure 4.2.

MOOSE implements the entities with the class `MSEAbstractEntity` and associations with `MSEAbstractAssociation`. Via the CCNode and CCEdge one can access the represented entity respectively the association and query it.

The most important class that connects Classifier with MOOSE is `MSEComputed-Group` respectively its subclass `ClassifierGroup`. The entire group hierarchy is shown in Figure A.3.

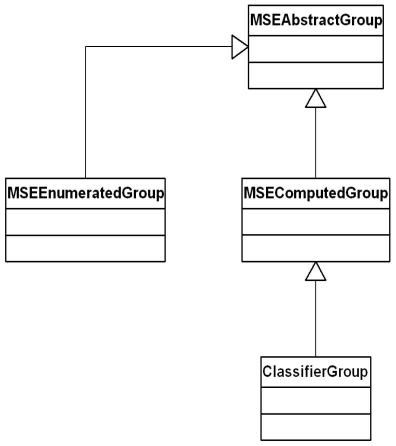


Figure A.3: Group hierarchy in MOOSE.

A.3 Connection to CodeCrawler

Classifier extends above all the CodeCrawler implementation. The main interface points are given through following classes:

CCItem. This class has one direct subclass called `CCCompositeItem`. Latter is used to describe the basic behavior of `CCCompositeNode` and `CCCompositeEdge` that represent the basic classes used for the logical as well as the graphical grouping.

CCItemPlugin. The classes `CCCompositeNodePlugin` and `CCCompositeEdgePlu-gin` subclass `CCCompositeItemPlugin`. The plugin class is responsible for the updating of class names in relation with the mouse movements on the CodeCrawler window.

CCLineFigure. For the displaying of composite edges we make use of the class `CCLineFigure`. This class is responsible for the visualisation of the line figures that represent edges. It implements the displaying methods of the edge figures. We do not subclass `CCLineFigure` because we do additional behavior.

CodeCrawler. This is the main class of CodeCrawler. It subclasses the class *DrawingEditor* from the HotDraw framework. Similarly, Classifier extends CodeCrawler. Classifier assumes the graphical grouping of nodes and edges.

For further information about CodeCrawler refer to [Lan99].

A.4 HotDraw

HotDraw is a two-dimensional graphics framework for structured drawing editors that was implemented in Smalltalk by John Michael Brant [Bra95].

A HotDraw application edits drawings that are made up of figures. Figures are graphics elements such as lines, boxes or text and they can represent other objects. A drawing editor built from HotDraw – *e.g.*, 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.

A.4.1 Connection to HotDraw

HotDraw uses the model-view-controller paradigm. The connection to HotDraw is given by:

CompositeFigure

GroupedFigure is implemented as direct subclass of CompositeFigure. This class is responsible for the visualisation of the rectangle figures that represent a group node. It implements displaying methods of the node figures.

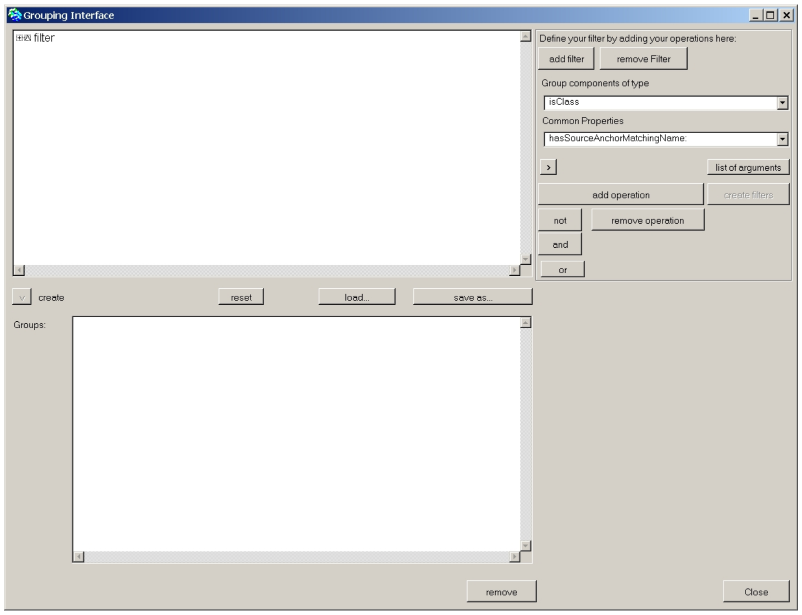
A.5 Most Important Classes

In this section we present the important classes in Classifier. We basically implement two graphical user interfaces that manage the group creation and the group visualisation.

Classifier. This is the main application class. It is a subclass of `CodeCrawler`, the main application class of the program CodeCrawler (see section A.3). Classifier basically adds the functionality of grouping, in particular aggregating, collapsing, and expanding.

ClassifierDrawing. The correspondant CodeCrawler class is named CCDrawing. ClassifierDrawing returns some supplemental information on displayed nodes and edges.

AbstractGroupUI. This is the class that describes the user interface *GroupGenerator* that enables generating groups. This is done by creating first a filter whose application results in a collection of entities. With the filter the user can generate a group, *e.g.*, an abstract MOOSE entity, that will contain the resulting entities. Here, the group only exists logically. The groups can be renamed or removed from the group list.


Grouping Interface

Define your filter by adding your operations here:

Group components of type: isClass

Common Properties: hasSourceAnchorMatchingName:

Figure A.4: AbstractGroupUI, a group generator interface. It allows to define groups by choosing the selection criteria.

GroupUI. The graphical group handling can be done with the interface called *GroupManager*. It is implemented by the class `GroupUI`. The GroupManager represents the interface for handling the groups in the CodeCrawler window.

All functions that can be applied from here are concerned with the graphical representation of the underlying metamodel. The functions are restricted to *selection, color, group creation* – collapsing or aggregation.

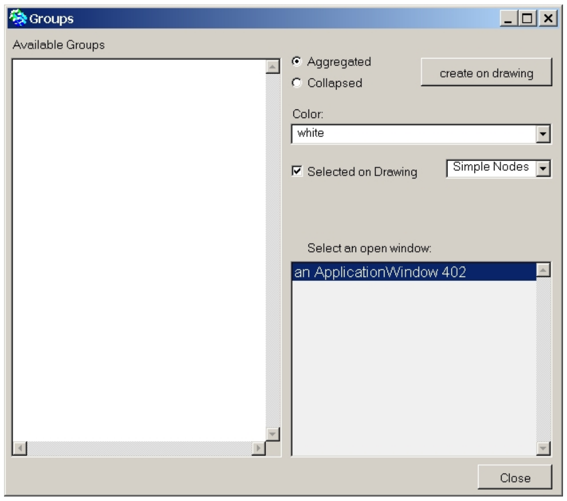


Figure A.5: GroupUI, a group manager. This window enables the interaction with graph view in the CodeCrawler window.

Operation. The class `Operation` implements filters. It contains an attribute called `suboperations` that contains other filters, each expressing an entity attribute. When a group is created the filters produce a query using the MooseQueries application described in 3.3.1. The complete Operation structure is represented in Figure A.6.

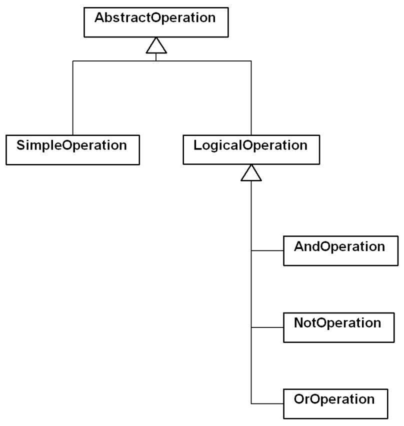


Figure A.6: The Operation hierarchy. Filters are implemented as Operation. The implementation corresponds to the composite pattern.

Bibliography

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison Wesley, 2000.

[Bra95] John Brant. Hotdraw. Master's thesis, University of Illinois at Urbana-Chanpaign, 1995.

[Cas98] Eduardo Casais. Re-engineering object-oriented legacy systems. *Journal of Object-Oriented Programming*, 10(8):45–52, January 1998.

[CC90] Elliot J. Chikofsky and James H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[CK91] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, pages 197–211, November 1991. Published as Proceedings OOPSLA '91, ACM SIGPLAN Notices, volume 26, number 11.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CM93] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[CMR92] Mariano P. Consens, Alberto O. Mendelzon, and Arthur G. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[CY91] Peter Coad and Edward Yourdon. *Object Oriented Analysis.* Prentice-Hall, 2nd edition, 1991.

[DD99] Stéphane Ducasse and Serge Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook.* University of Bern, October 1999. See http://www.iam.unibe.ch/~famoos/handbook.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings WCRE'99 (6th Working Conference on Reverse Engineering).* IEEE, October 1999.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In Bernhard Rumpe, editor, *Proceedings UML'99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, Kaiserslautern, Germany, October 1999. Springer-Verlag.

[Duc01] Stéphane Ducasse. Reengineering object-oriented applications. Technical report, Université Pierre et Marie Curie (Paris 6), 2001. Habilitation.

[ESEE92] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[FP96] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* International Thomson Computer Press, London, UK, second edition, 1996.

[Gla02] Robert L. Glass. Sorting out software complexity. *Communications of the ACM*, 45(11):19–21, November 2002.

[KC98] Rick Kazman and S. Jeromy Carriere. View extraction and view fusion in architectural understanding. In *Proceedings of the 5th International Conference on Software Reuse*, Victoria, B.C., 1998.

[KC99] Rick Kazman and S. J. Carriere. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, April 1999.

[Lan99] Michele Lanza. Combining metrics and graphs for object oriented reverse engineering. Diploma thesis, University of Bern, October 1999.

[Lan03] Michele Lanza. Codecrawler - lessons learned in building a software visualization tool. In *CSMR 2003 Proceedings (European Conference on Software Maintenance and Reengineering)*, 2003.

[LD01] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In *Proceedings of OOPSLA 2001*, pages 300–311, 2001.

[LK94] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[Mue86] Hausi A. Mueller. *Rigi - A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of International Conference on Software Engineering*, 1994.

[Riv00] Claudio Riva. Reverse architecting: an industrial experience report. In *Proceedings of WCRE'00*. IEEE Computer Society, 2000.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

[SDN02] Stphane Ducasse Serge Demeyer and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of the 1995 International Conference on Software Maintenance*, 1995.

[SRT92] Mehmet A. Orgun Scott R. Tilley, Hausi A. Mller. Documenting software systems with views. In *Proceedings of the 10th International Conference on Systems Documentation*. ACM, 1992.

[Sta90] John T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, September 1990.

[Ste01] Lukas Steiger. Recovering the evolution of object oriented software systems using a flexible query engine. Diploma thesis, University of Bern, June 2001.

[Tah00] Ladan Tahvildari. Quality-driven object-oriented software reengineering. Diploma thesis, University of Waterloo, 2000.

[Tic01] Sander Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[TWSM94] Scott R. Tilley, Kenny Wong, Margaret-Anne D. Storey, and Hausi A. Müller. Programmable reverse enginnering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, 1994.