TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Architecture Reconstruction of Industrial Object-Oriented Software

A Case Study

by
A. Wierda

Supervisors:

Dr. L.J.A.M. Somers
Ir. H.M.J.M. Dortmans

Eindhoven, August 2005

Disclaimer

The writer was enabled by Océ-Technologies B.V. to perform research that partly forms the basis for this report. Océ-Technologies B.V. does not accept responsibility for the accuracy of the data, opinions and conclusions mentioned in this report, which are fully for the account of the writer.

A Abstract

The architecture of a software system represents a blueprint of the system. Having an up to date architecture description is an important prerequisite for software maintenance, which represents a large portion of a software project's total costs. In practice however such a description is often not available.

This thesis focuses on reconstructing the as-built architecture of large object-oriented systems from their source code. In literature several types of methods for this are described, which are summarised in this thesis. After an extensive literature study this thesis describes two case studies that apply the two most prominent automatic methods, pattern detection and architectural clustering, to a large industrial object-oriented system called the Océ Controller.

Pattern detection methods discover recurring solutions in a system's implementation, for example design patterns in object-oriented source code. Usually this is done with a pattern library. This has the disadvantage that the precise implementation of the patterns must be known in advance. The method used in our first case study does not have this disadvantage. It uses a mathematical technique called formal concept analysis and is applied to find structural patterns in two subsystems of the Océ Controller. The case study shows that it is possible to detect frequently used structural design constructs without upfront knowledge. However, even the detection of relatively simple patterns in relatively small pieces of software takes a lot of computing time. Since this is due to the complexity of the applied algorithms, applying the method to large software systems like the complete Océ Controller is not practical. They can be applied to its subsystems though, which are about five to ten percent of its size.

Architectural clustering uses mathematical clustering to group closely related source code elements into higher-level abstractions, usually for procedural software. In our second case study clustering is used to group classes of an object-oriented system into subsystems. The clustering process is based on the structural relations between the classes. More precisely, on associations, generalizations and dependencies. The clustering is performed with a third party tool called Bunch, which is a clustering tool for procedural software that, to our knowledge, has not been previously applied to object-oriented software. In the case study the clustering method has been used to reconstruct the architecture of the last two versions of the Océ Controller. Compared to other clustering methods the results come relatively close to the result of a manual reconstruction. However, some manual refinement is still needed. Performance wise the clustering takes a significant amount of time, but not too much to make it unpractical.

The clustering is based on the structural relations between the classes. To our knowledge no work is published on the importance of the different relationship-types for the clustering result and how best to incorporate this information in the clustering process. We experimented with different combinations of relationships and different ways to use this information in the clustering process. The results clearly show that dependency relations are vital to achieve good clusterings.

To our knowledge clustering methods reported in literature are always based on information of a single version. If multiple versions of a system have been released this leaves a lot of information unused. In our case study we based the clustering on information from multiple versions and compared the result to that obtained when basing the clustering on a single version. We experimented with several combinations of versions. If the clustering was based on relations that were present in both the reconstructed *and* the first version this led to a significantly better clustering result compared to that obtained when using only information from the reconstructed version.

B Preface

In the summer of 2004 I was discussing the possibilities for a graduation assignment with various people at Océ. Most assignments were in the field of improving an existing system, or determining how a system has been implemented. The latter was necessary to facilitate the further evolution of these systems. Typically, the documentation was not up to date with the implementation, and the systems were relatively large and complex. Since an up to date description of a system's essential parts is a prerequisite for successful evolution, we decided to focus on the reconstruction of software architectures.

A software architecture represents a blueprint of a software system that describes its fundamental organisation. Architecture reconstruction recovers lost parts of this blueprint. The term "architecture" raises the analogy to construction architectures of, for example, buildings. When buildings require extensions or renovations, the builders need to understand the existing structure. For example they need to know the locations of pipes and support beams, and the strength of foundations. Otherwise they risk damaging the existing building during their work. Therefore the builders study the existing structure before making any changes. Architecture reconstruction of software systems does the same of software. It recovers the architecture of an existing software system with the aim to facilitate future changes.

During the project we focussed on the reconstruction of the architecture of large software systems. The size of these systems places and extra burden on the reconstruction process. In two case studies we applied two automatic architecture reconstruction techniques to a large software system. The two techniques have in common that they reduce the amount of information given to developers, yet preserve the essence of the architecture. Initially we focussed on the detection of design patterns that are frequently used in the implementation. Knowledge of these design patterns can be used to understand systems more efficiently. After some disappointing results that were mainly due to the inherent complexity of the used algorithms, we shifted our focus to another frequently used reconstruction technique called architectural clustering. This technique groups the structural elements of a system into abstract entities. This is necessary to prevent developers from being overwhelmed by the sheer size of the system. The grouping process heuristically optimises generally accepted design criteria, similar to what a human would do.

I want to express my gratitude to my advisor Eric Dortmans and my supervisor Lou Somers for their support and constructive comments during the assignment. Also, I want to thank Michel Chaudron and Teade Punter for taking place in my examination committee. Additionally, I want to thank Rob Kersemakers and Wim Couwenberg for their constructive comments on an early version of this document.
Despite their busy work ten designers and architects made time to participate in an experiment in which the architecture of a small program is reconstructed. I want to thank Jacques Bergmans, Patrick Vestjens, Marvin Brunner, Erwin van der Linden, Wim Couwenberg, Michel de Groot, Peter Nacken, Jantinus Woering, Erik Scheppink and Eric Dortmans for their participation.
This thesis completes my study at Eindhoven Technical University. I want to thank my family, my friends, and my colleagues at Océ Research & Development for their help and support during this period.

Andreas Wierda

Venlo, June 2005

C Table of contents

D List of abbreviations

ANSI	American National Standards Institute
API	Application Programming Interface
COTS	Common Of The Shelf
FAMOOS	Framework-based Approach for Mastering Object-Oriented Software
FAMIX	FAMoos Information eXchange Model
FCA	Formal Concept Analysis
FTP	File Transfer Protocol
HTTP	HyperText Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
KLOC	Kilo-Lines Of Code
MDG	Module Dependency Graph
RIP	Raster Image Processing
RSF	Rigi Standard Format
SMB	Server Message Block
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
STBC	Source-Tree Based Clustering
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modelling Language
XML	eXtensible Mark-up Language
XSLT	eXtensible Stylesheet Language Transformations

E List of figures

F List of tables

1 Introduction

This chapter discusses the assignment, the system that is analyzed in the two case studies described in this thesis, and the structure of this document.

1.1 Context

The subject system for the two case studies described in this thesis is an Océ Controller. Such a controller consists of general-purpose hardware on which software of Océ and third parties is running. Its main task is to control (physical) devices such as a print- and scan-engine, and act as an intermediate between them and the customer network. This is illustrated in Figure 1, where the blocks represent systems in the environment of the controller and the lines some physical connection between two or more systems.

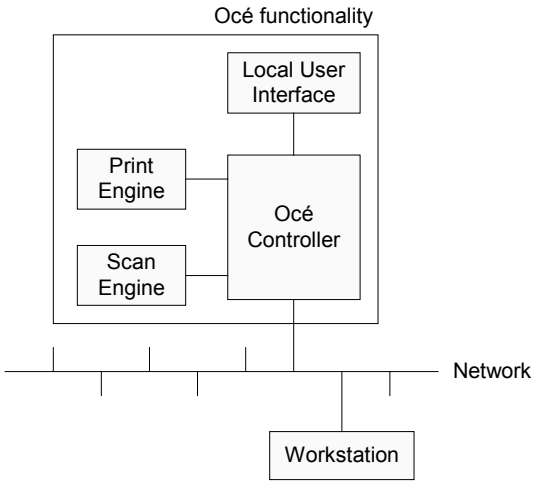


Figure 1: Océ Controller context

The Océ Controller handles the workflow within an Océ multi-functional device. Typical tasks of the Océ Controller are:

- Reception of postscript files the user sent from a workstation and the conversion of these files to a bitmap format the Print Engine accepts. In this conversion the settings specified by the user are taken into account.
- Initialise the Scan and Print Engine for a copy-job and control the workflow during the execution of the job.
- Provide job- and queue-management functions to the Local User Interface.
- Handling network connectivity to the network of the customer. For this task the controller supports various protocols, including TCP/IP, SMB, FTP, HTTP and SNMP.
- Provide a remote user interface (web based).

1.2 Software characteristics

The software running on the Océ Controller has been written in multiple programming languages, but mostly in C++ [Stroustrup, 1997]. An as-designed architecture is available, but it is not complete and large parts of the architecture documentation are not consistent with the implementation [Lange, 2003].

Table 1 shows the characteristics of the Océ Controller and two of its subsystems, Grizzly and RIP Worker. Because of performance limitations it was not feasible to apply the design pattern detection described in chapter 5 to the complete Controller. Instead, it has been applied to these two subsystems. The Grizzly subsystem [Delnooz and Vrijnsen, 2003] provides a framework for prototyping on the Océ Controller. The RIP Worker subsystem [DVRIP, 2002] transforms Postscript files into printable bitmaps, taking the print-settings the user specified into account (“ripping”).

The architectural-clustering described in chapter 6 has been applied to the complete Océ Controller. Because this process uses input from multiple versions, Table 1 gives the statistics for the first (1) and the last (8a) version of the Océ Controller.

	Controller (v. 1)	Controller (v. 8a)	Grizzly	RIP Worker
Classes	1.545	2.661	234	108
Header and source files	4.378	7.549	268	334
Functions	21.711	40.449	2.037	1.857
Lines of source code (*1000)	453	932	35	37
Executable statements (*1000)	167	366	18	16

Table 1: Software characteristics

1.3 Assignment

The research questions that led to this thesis were:

1. Which methods are available for software architecture reconstruction?
2. Can these methods be used to reconstruct the architecture of the Océ Controller?
3. How good are the results?
4. How can these methods be improved?

[Kersemakers, 2005] reconstructs the architecture of the Océ Controller by detecting instances of architectural styles and design patterns in the source code. The implemented approach uses a pattern library that specifies the patterns searched for. The work described in this thesis builds on this work and complements it.

The Océ Controller suffers from several of the typical legacy problems discussed in paragraph 2.2. In discussions with developers, issues like limited understanding of the system, obsolete documentation, unexpected dependencies and code smells are mentioned. The size of the system implies two important requirements for the reconstruction methods:

- They must work automatic or semi-automatic, and not completely manual.
- They must be scalable enough to handle large systems like the Océ Controller.

Furthermore, the fact that multiple programming languages are used in the Océ Controller implies the requirement that the methods must be language independent.

To answer the above questions the following research approach was used. We started with a literature study on architecture reconstruction. From this we concluded that pattern detection and architectural clustering seemed suitable for the recovery of the architecture of the Océ Controller. To validate this assumption both methods have been applied to the Océ Controller, leading to the conclusions in chapter 8.

1.4 Report structure

This report is structured as follows. After the introduction in chapter 1, chapter 2 elaborates on the background of architecture reconstruction. Architecture reconstruction is a form of reverse engineering that aims to describe a system in terms of higher-level abstractions beyond those obtained from the source-code itself. Reverse engineering techniques are often applied to legacy software, to reconstruct lost knowledge in order to ease maintenance.

Chapter 3 describes several general approaches for architecture reconstruction reported in literature. The detection of design patterns and architectural clustering are the two most prominent approaches for automatic architecture reconstruction.

Chapter 4 describes pattern detection methods reported in literature, after which chapter 5 describes a case study that applies this theory. It attempts to reconstruct an architectural view of the Océ Controller by detecting frequently used design constructs in the source code. Unlike many other methods for detecting design pattern instances in source code, the method used in this case study does not require upfront knowledge on the expected patterns.

Chapter 6 describes architectural clustering methods reported in literature. This theory is applied in chapter 7, where a case study that uses clustering to reconstruct an architectural view of the Océ Controller from its source code is described. Clustering techniques discover a natural ordering of a set of elements. In the described case study, the elements are classes and the ordering is based on the structural relations between the classes.

Finally, chapter 8 draws conclusions from the two case studies and describes future work.

This thesis comes with several appendices. Appendix 1 lists the references. The next three appendices describe the output formats of the pattern detection prototype's modules. This prototype was used during the case study described in chapter 5.
Appendix 5 describes decompositions of a small sample application that are created by experienced architects. These are the result of the experiment described in paragraph 7.3.2. Appendix 6 and Appendix 7 describe results of the architectural clustering case study discussed in chapter 6. The first describes the results for two subsystems of the Océ Controller and the second for the complete Océ Controller.

2 Problem background

This chapter discusses the role of architecture reconstruction in the software lifecycle. In this context subjects like software maintenance, legacy software, reverse engineering and software architecture are discussed.

2.1 Software maintenance

This paragraph discusses software maintenance, its costs, problems, and its role in the software lifecycle.

2.1.1 Software maintenance in the software lifecycle

Throughout the software lifecycle several processes can be distinguished. In the primary process[1] of the software lifecycle [ISO 12207] distinguishes among others the following sub-processes:

1. Development
2. Operation
3. Maintenance

Regardless of the precise lifecycle model that is used, maintenance is an integral part of the software lifecycle. Both in scientific and software engineering literature there is consensus that maintenance accounts for a large portion of the total costs of a software project. Based on case studies and research in industrial projects it is estimated that 50-90% of the total software costs is spent on maintenance. Table 2 lists the results of several studies on this subject. References marked with * are cited by [Koskinen, 2004]. Although definition differences make the figures hard to compare, it is clear that software maintenance costs represent a very large part of the total software costs.

[Erlikh, 2000]	software costs for system maintenance and evolution / total software costs	85-90%
[Chikovsky and Cross, 1990]	software maintenance costs / total life-cycle costs	50-90%
[Moad, 1990]*	software costs for system maintenance and evolution / total software costs	>90%
[McKee, 1984]	software maintenance effort / total available software engineering effort.	65-75%
[Horowitz and Munson, 1984]	software maintenance costs / total life-cycle costs	72%
[Lientz and Swanson, 1981]*	staff time spent on maintenance / total time	>50%
[Nosek and Palvia, 1980]	software costs for system maintenance and evolution / total software costs	60-80%
[Sommerville, 2004]	software maintenance costs / total software engineering effort for long-lived systems	75-80%
[Zelkowitz et al, 1979]*	maintenance costs / total software costs	67%

Table 2: Proportional software maintenance costs

2.1.2 Definition

Software maintenance not only comprises of correcting faults, but also of performing adaptations to keep the software fit for use. The IEEE definition of software maintenance is *"the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* [IEEE 610].

[1] Besides the primary process [ISO/IEC 12207] also distinguishes supporting and organizational processes.

In software maintenance literature three types of software maintenance are commonly distinguished [Lientz and Swanson, 1981], [Swanson and Chapin, 1995]:

- **Corrective maintenance** is concerned with fixing reported errors in the software. Empirical studies show that this accounts for 17% of the maintenance costs[2].
- **Adaptive maintenance** is concerned with adapting the software to changes in the operating environment (e.g. platform and operating system changes). Such changes can violate assumptions embedded in the design of the software, leading to unexpected behaviour. According to Lehman's law of continuing change these type of adaptations are unavoidable for software in real world applications: *"an E-Type[3] program that is used must be continually adapted else it becomes progressively less satisfactory"* [Lehman, 1996]. Adaptive maintenance accounts for 18% of the total maintenance costs[2].
- **Perfective maintenance** is concerned with implementing new functional or non-functional requirements. This accounts for 65% of the maintenance effort[2]. Lehman's law of continuing growth states that *"the functional content of a program must be continually increased to maintain user satisfaction over its lifetime"* [Lehman, 1996]. [Turski, 1981] pointed out that this is in fact an abuse of the term maintenance: the addition of a new wing to a building is never called maintenance of that building. Therefore the term evolution is actually more appropriate [Koschke, 2000].

[Baldo, 1995] identifies two other types of software maintenance; preventive and structural maintenance. Preventive maintenance is concerned with activities that prolong the effectiveness and reliability of the system. Structural maintenance is the modification of software with the aim to improve its future maintainability.

2.1.3 Program understanding during maintenance

From the above it is clear that software maintenance is an important part of the software lifecycle. The constant need for changes identified by Lehman's law of continuing change leads to continuous modification of a program throughout its lifetime. During this process the interactions and dependencies between system elements increase in an unstructured way, increasing the system entropy[4]. Lehman's law of increasing complexity states that "*a program's complexity increases as it evolves, unless work is done to maintain or reduce it… If this growth in complexity is not constrained the program will take progressively more effort to maintain"* [Lehman, 1996].

Developers performing maintenance usually start by understanding the problem and the concerned parts of the software. The gradual deterioration of the software's internal structure Lehman identified makes this increasingly difficult. The reason for this is that it becomes more and more difficult to understand the structure of a program. [Fjeldstadt and Hamlen, 1984] showed that maintenance programmers performing adaptive or perfective maintenance spend 47% of their time studying the program source code and the associated documentation. When performing corrective maintenance this increases to 62%. This means that a reduction of the effort needed to understand the internal structure of software directly affects the total costs of the project.

2.2 Legacy software

This paragraph discusses a specific type of software called "legacy software". This is old software that is still used and maintained. The "legacy" aspect makes understanding and maintaining it more difficult than with non-legacy software.

2.2.1 Definition

Webster's Dictionary defines legacy as *"anything handed down, or as from, an ancestor"*. In the context of software, legacy software refers to a piece of inherited software. In practice only valuable software is inherited, so legacy software refers to valuable software that has been inherited [Demeyer et al, 2004].

[2] [Sommerville, 2004].
[3] An E-Type program denotes software that solves problems in the real world.
[4] A system's entropy is the amount of disorder in it.

2.2.2 Legacy problems

The mere fact that a piece of software is classified as legacy does not necessarily imply problems with it. The difficulties emerge when the software can no longer be maintained. However, in literature the term legacy software frequently implies the presence of maintenance difficulties.

[Demeyer et al, 2004] describe symptoms that indicate maintenance difficulties, now or in the near future:

- **Obsolete or no documentation.** The absence of up to date documentation is a clear warning sign that maintenance difficulties will arise. When systems have undergone many changes the documentation often no longer reflects the actual implementation.
- **Missing unit- and system-tests.** If unit- and system-tests are not available further evolution of the system is very risky because it is not possible to determine if changes broke existing functionality.
- **Limited understanding of the system.** This can concern the overall structure of the system or important implementation details. Both can seriously hinder system evolution. Reasons for a limited understanding of the system can be the departure of the original developers or users, or simply because important details have been forgotten. In combination with a lack of up to date documentation and missing tests, this can lead to a rapid decline of the system's quality when it evolves.
- **Too much time needed to make simple changes.** If simple changes require disproportional development effort, complex changes are likely to be unfeasible. This means the system can no longer evolve to keep up with customer demands.
- **Unexpected dependencies.** When a change is made, for instance to fix a bug, the software breaks in unexpected places. If this happens frequently the architecture might not be able to accommodate future needs. This is a clear warning that the structure of the system is not fully understood.
- **Too long before changes are available in the field.** The process of adapting the system to changing needs stalls somewhere. It might be that it takes too long to decide which changes will be made, their implementation takes too long, or transferring them to operational use takes too long.
- **Big build times.** If build times grow more than the system size this indicates that the internal complexity of the system has increased to a level where compiler tools can no longer do their work efficiently. This indicates that the architecture of the system has deteriorated.
- **Difficulties separating products.** This can happen in situations where different clients or products use a system. Difficulties separating releases from each other indicate that the architecture is not able to accommodate these changes anymore.
- **Duplicate code.** Duplicate code is created routinely when developers need nearly identical code in multiple places and there is no time for a structural solution. If the common parts of duplicate code are not refactored into suitable abstractions the duplicated code remains in place. Then changes to the system lead to the same code being updated in multiple places, quickly increasing the effort needed for maintenance.
- **Code smells.** Duplicated code is an example of a code smell. Other examples are long method bodies, big classes, long parameter lists et cetera. Code smells can indicate that a system has been expanded repeatedly without adjusting its internal structure.

Note that it is not so much the absolute age of software that makes it legacy software. Instead, the amount of changes to the software and its environment since the software was created are the determining factors [Demeyer et al, 2004]. Examples of such changes are new development methods, design paradigm shifts and project staffing changes. Legacy software is software that has undergone many of these changes without being refactored properly.

2.2.3 Rebuilding legacy software from scratch

One might argue that legacy problems can simply be solved by throwing the software away and rebuilding it from scratch. [Ducasse, 2003] and [Finnigan et al, 1997] mention several reasons why this is not possible:

- **Size**: Legacy systems are often very large and implement a lot of functionality. Much of the knowledge that drove their evolution has been lost and it is often impossible to overview the complete system.
- **Mature**: Customers using the software are generally quite satisfied with it until they need a new feature. However, they do not want to pay for a complete re-development.
- **Value**: Legacy systems constitute a significant asset for their owners and are often an important source of income.

These factors, and the risks of failure associated with every new project, cause organisations to prolong the life of legacy software. Taking this one step further; in some domains of industry increasing functional demands and decreasing time-to-market make it impossible to develop each new product from scratch. According to [Krikhaar et al, 1999] this is already the case in the domain of high-volume electronics.

2.2.4 Object-oriented legacy software

Originally the term legacy software was used for programs written in languages like assembler, Cobol or Fortran. However legacy problems are not constrained to specific types of languages. Changing environments and requirements also affect object-oriented software. [Ducasse, 2003] mentions several projects where object-oriented legacy systems are reengineered. It turns out that legacy software even exists in relatively young programming languages such as Java [Java, 2005].

[Macl and Havanas, 1990] and [Kiran et al, 1997] both describe empirical studies comparing the maintenance effort for object-oriented software with that of traditional procedural software. They both conclude that for equivalent changes software maintenance in object-oriented software takes less effort than in traditional procedural software. However, this has had an interesting side effect. Based on an empirical study described in [Dekleva, 1992], [Grass, 1998] states that modern development methods lead to:

1. more reliable software,
2. less frequent software repair and
3. *more total maintenance time*.

These statements are not contradictionary, neither with themselves nor with the conclusions of [Macl and Havanas, 1990] and [Kiran et al, 1997]. Because modern methods make software better maintainable and easier to enhance, software is changed more often [Grass, 1998]. Following the classification described in paragraph 2.1, object-orientation has led to a reduction of corrective maintenance costs and an increase in adaptive and perfective maintenance costs.

This increasing rate of change causes object-oriented software to become legacy much sooner than non-object-oriented software. [Demeyer et al, 2004] states that *"it is not the age that turns a piece of software into a legacy system, but the rate at which it has been developed and adapted without having been reengineered"*. [Casais, 1998] pointed out that reengineering is actually an essential element of iterative development processes.

Besides the problems normally encountered with legacy software, inheritance and polymorphism cause new problems [Wilde and Huiit, 1992]:

- Many traditional maintenance tools depend on dependency tracing. Polymorphism and dynamic binding makes this much more difficult, especially in dynamically typed languages.
- The use of inheritance and incremental class definitions, together with the dynamic nature of "self" and "this", make understanding classes more difficult. The reason for this is that the ancestors of a class much also be examined when trying to trace which method will be executed.

- Object-oriented programming styles promote the use of many relatively small classes. This leads to domain models and functionality being spread over different classes, making it difficult to locate their implementation.
- Understanding the high level structure of object-oriented systems is more difficult. In non-object-oriented systems the calling hierarchy is often used as a starting point, but in object-oriented systems this has several disadvantages. First, dynamic binding makes it difficult to compute. Second, there may not be a real "main" method, so it is unclear where to start [Lanza, 2003a]. Third, in such a call graph the grouping of methods into objects is lost, which is usually an important design property.

By the end of the 1990's early industrial adopters of object-orientation were already facing problems with the evolution of some large object-oriented systems [Casais, 1998]. Besides the difficulties inherent to the nature of object-oriented software, misuse and abuse of object-oriented features are common problems too. Some examples are [Ducasse, 2003]:

- **Misuse of inheritance**: the use of inheritance to achieve composition, or simply code reuse, instead of for defining abstractions. In his “Is-a” rule for inheritance [Meyer, 1998] states that a class B may only inherit from a class A if an argument can be made that every instance of B can also be viewed as an instance of A. If two classes are related by a “has-a” relation a client-server coupling must be used. In many cases where inheritance is applicable, a client-server coupling could also be used. In such cases inheritance must only be used if polymorphism is used also.
- **Missing inheritance**: duplicated code and case statements where inheritance and dynamic binding would be more appropriate.
- **Misplaced operations**: this can be caused by unexploited cohesion, or simply because operations are placed in the wrong class.
- **Violation of encapsulation**: explicit typecasts, C++ friend classes.
- **Class abuse**: lack of inner-class cohesion and the use of classes as namespaces.

2.3 Reverse engineering

Architecture reconstruction is a form of reverse engineering. This paragraph defines reverse engineering and discusses several important types of reverse engineering, including architecture reconstruction. Finally, several reverse engineering methods are introduced.

2.3.1 Definition

[Chikovsky and Cross, 1990] define reverse engineering as *"the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction"*. Reverse engineering does not involve changing a system or producing new systems based on existing systems, but is concerned with understanding a system.

The counterpart of reverse engineering is 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"* [Chikovsky and Cross, 1990].

2.3.2 Reverse engineering goals

The main goal of reverse engineering is to increase the understanding of a system. Biggerstaff considers program understanding *"a common, sometimes hidden part of many activities scattered throughout the software lifecycle"*. Any developer working on software that is new to him or her spends a lot of time trying to understand it. According to [Fjeldstadt and Hamlen, 1984] a maintenance programmer performing adaptive or perfective maintenance spends 47% studying the source code and the documentation. When performing corrective maintenance this increases to 62%.

2.3.3 Reverse engineering types

Redocumentation and design recovery are two widely known subareas of reverse engineering that both produce artefacts that help to understand a system. They can be distinguished by the abstraction level of their results and the used knowledge sources.

Redocumentation usually uses source code as input and produces equivalent representations of it within the same relative abstraction level. The results are often considered alternative views of the software.

Design recovery combines different knowledge sources to produce abstractions of a subject system [Biggerstaff, 1989]. The goal is to *"obtain meaningful higher-level abstractions beyond those obtained directly from the source code itself"* [Chikovsky and Cross, 1990].

[Mansurov and Campara, 2003] draw the analogy with archaeology. They use the term *architecture excavation* to refer to design recovery. [Koschke, 2000] prefers the term *architecture recovery* to refer to this process because it is considered more general.

[Kazman et al, 2001] use the term *architecture reconstruction* to refer to the process of extracting the as-built architecture of an existing system. Since this clearly involves obtaining higher-level abstractions than those defined in the source code, the difference, if any, with design recovery has to be defined. [Buschmann et al, 1999] distinguish architecture and design based on abstraction level and scope. An *architecture* uses higher-level abstractions than a design and describes a complete system, whereas a *design* describes the interior of specific subsystems of the architecture. Note that this does not define a precise boundary between architecture and design.

Considering the above definitions we will use the term *design recovery* for methods that can be used to reconstruct designs, but not architectures. Such methods construct abstractions at a higher level than that obtained directly from the source code, but do not reach the abstraction level used in the architecture. *Architecture reconstruction* will refer to methods that reconstruct architectures, producing elements of the associated abstraction level.

2.3.4 Reverse engineering approaches

There are many reverse engineering tools and techniques. They use a variety of information sources, including the ones below [Demeyer et al, 2004]:

- Existing documentation, including manuals.
- Source code and its directory structure.
- Test runs of the software and execution traces.
- Interviews with users and developers.
- Test cases.
- Version history information.

In practice, source code is the most important information source for reverse engineering [Trevors and Godfrey, 2002], [Buckley, 1989]. The latter publication identifies two reasons for this:

- Design documentation does not match the implementation. This gets worse as the system evolves because its structure deteriorates.
- Design documentation is not designed for maintenance but for forward engineering.

[Nelson, 1996] classified reverse engineering techniques into three distinct approaches, based on the type of input they use. This classification distinguishes methods based directly on the source code, methods based on an abstract graph representation of the source code and methods based on executions:

- **Textual, lexical and syntactic analysis methods** are based directly on the source code. This includes producing cross-reference listings, abstract syntax trees and control flow graphs.
- **Graph-based methods** are based on an abstract graph that represents the source code. Methods such as control- and data flow-analysis, and program dependence charts produce graphs considering the source code from a certain perspective. Slicing methods extract the part of the source code that affects a certain variable at a certain point in the source code. Pattern recognition methods search for recurring programming patterns. Clustering methods impose a new ordering on the software by partitioning the program graph into disjoint parts while optimising design trade-offs.

- **Execution and testing methods** are based on information obtained from full, partial or simulated executions. This includes profiling methods for analysing the performance, and testing methods for estimating the degree of correctness[5] of the software. Abstract interpretation is a method that performs static testing by simulating the software's dynamic behaviour. Finally, partial analysis can be used to isolate parts of the software for analysis purposes.

This thesis describes two case studies that use graph-based reverse engineering methods. The first is based on detecting recurring design patterns. The second uses clustering techniques to find suitable system decompositions. According to [Sartipi and Kontogiannis, 2003] these two approaches are the most prominent methods for semi-automatic architecture reconstruction.

Pattern-based techniques are used to find common structures or solutions in the architecture of software systems. When considering the program as an abstract graph this amounts to finding frequently occurring subgraphs. This can be done by matching a library of known patterns with the program graph, or with algorithms that find frequently occurring subgraphs without a priori knowledge. In both cases the end result is a list of frequently used constructs, each with a list of instances. This result can be used to understand the system faster by abstracting common constructs.

Clustering-based techniques impose an ordering on the system by grouping closely related program entities into subsystems. The algorithms usually start with an abstract graph representing the structure of the program, for example with the nodes representing classes and the edges inter-class relationships. Some similarity measure is then used to find groups of similar or closely related classes, which are grouped into subsystems. This is repeated until an optimal decomposition is found. The end result can be browsed top-down, helping to understand the complete program.

In the context of architecture recovery, pattern detection and clustering are two complementary approaches; the first finds common abstractions embedded in the system, but in practice never covers all entities in the system [Quilici, 1995]. The second classifies all entities in the system, but imposes a new ordering instead of some hidden ordering.

2.3.5 Software transformations

As stated, reverse engineering does not involve changing the software. Restructuring, reengineering and refactoring on the other hand do. [Chikovsky and Cross, 1990] define reengineering as *"the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form"*. This generally involves some form of reverse engineering, followed by forward engineering to implement the desired changes. This may also include implementing new requirements (perfective maintenance).

Restructuring improves the internal structure of the software, but does not involve implementing new requirements. According to Chikovsky and Cross, restructuring is *"the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behaviour (functionality and semantics)"*. Examples of restructuring are source code translations (e.g. to remove "goto" statements) but also design changes.

Restructuring refers to general source code translations. Restructuring in an object-oriented context is called refactoring [Demeyer et al, 2004]. [Fowler et al, 1999] define refactoring as *"the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure"*.

[5] Testing can never prove the correctness of a program, but it can be used to estimate the number of remaining errors. An example is described in [Ehrlich et al, 1990].

2.4 Software Architecture

Architecture reconstruction recovers an architecture from source code. This paragraph defines software architecture in general and methods to specify software architectures. Further, frequently used architectural blueprints called architectural styles are discussed.

2.4.1 Definition

As stated before, architecture reconstruction reconstructs the architecture of an existing system. Before discussing this in more detail it is important to understand what an architecture is. There is no generally accepted definition of software architecture, but there is no shortage of definitions either. [SEI, 2003] presents a set of definitions, of which some of the more popular ones are presented in this paragraph.

[IEEE 1471] defines architecture as *"the fundamental organization of a system embodied by its components, their relationships to each other and to the environment and the principles guiding its design and evolution"*. This definition captures the underlying elements of many definitions for the term architecture. The most important element is the need to understand and control the elements of the system that *"capture the system's utility, cost and risk"* [IEEE 1471]. Another important element are the design principles on which the system is based.

[Booch et al, 1999] define architecture as *"the set of significant decisions about:*

- *The organization of a software system.*
- *The selection of the structural elements and their interfaces by which the system is composed.*
- *Their behavior, as specified in the collaborations among those elements.*
- *The composition of these structural and behavioral elements into progressively larger subsystems.*
- *The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition"*.

[Bass et al, 2003] define the software architecture of a program or computing system as *"the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"*. "Externally visible" refers to the assumptions that can be made about the elements of the system. Examples of these are the provided services, performance, fault handling and resource usage. The definition explicitly mentions multiple structures. This indicates multiple views of the same system can exist that collectively form the architecture.

2.4.2 Architectural views

A software architecture serves many different stakeholders, each having different needs. For example system engineers, end-users, programmers and integrators all need different information.

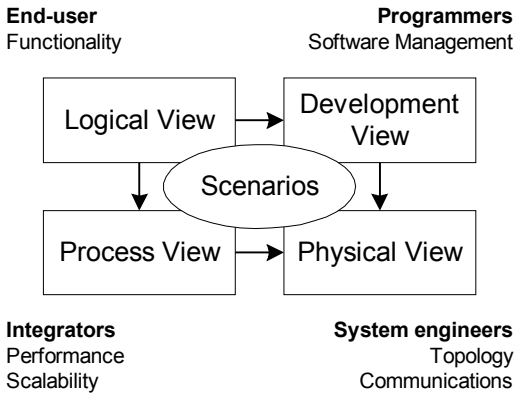


Figure 2: The "4+1" view model (from [Kruchten, 1995])

To prevent architectures from becoming large, cluttered diagrams with obscure boxes and lines [Kruchten, 1995] proposed the use of multiple views to describe architectures of software intensive systems. In Kruchten's 4+1 model the five views illustrated in Figure 2 are distinguished:

- **The logical view** primarily supports the functional requirements of the end-user. In this view the system is decomposed into a set of key abstractions that are taken from the problem domain. In case of an object-oriented architecture these are usually modeled as classes and objects, and exploit the principles of abstraction, encapsulation and inheritance. Besides modeling the functional requirements, this view also serves to identify common design elements and mechanisms.
- **The process view** describes the concurrency and synchronization aspects of the architecture, taking non-functional requirements concerning for example concurrency, performance, and availability into account. This view also describes how the elements of the logical view map to the process architecture, possibly at several abstraction levels.
- **The development view** guides the development process. It describes the mapping of the elements of the logical view to the software development environment. The software is packaged in small chunks -libraries or subsystems- that are organized in a hierarchical structure. Besides this hierarchical relationship the chunks are related by import and export relationships.
- **The physical view** describes how the elements of the logical, process and development views are mapped to the hardware. This mapping is mainly determined by non-functional requirements such as availability, reliability, performance and scalability.

The scenarios represent the "+1" in "4+1". This view describes the most important functional scenarios to demonstrate how the other four views work together. Like the logical view, the scenarios are an abstraction of the functional requirements. This view is redundant with respect to the other views. During the architecture construction process it serves as a driver to discover the architectural elements in the other views. When the architecture is completed it is used for validation and illustration.

The "4+1" view model is part of the Unified Modelling Language (UML, [Booch et al, 1999]). [Hofmeister et al, 1999] describe an alternative model that provides better support for modelling dynamic aspects of the architecture. For more information on this model the interested reader is referred to [Hofmeister et al, 1999].

2.4.3 Architectural styles and design patterns

An architecture is usually based on knowledge and experience of the architects that constructed it. *Patterns* provide proven solutions to recurring design problems in a specific context. [Alexander, 1979] first described common problem/solution pairs in urban architecture. [Gamma et al, 1995] extend this idea to object-oriented software development. They describe a set of 23 design patterns in a common format. This format describes the design problem, its context, appropriate terminology, one or more solutions, and their properties.
In practice the specific abstractions of data, function and interconnections introduced by the patterns serve as abstractions of common coding constructs [Beck et al, 1996].

Design patterns are believed to be beneficial in several ways [Beck et al, 1996], [Gamma et al, 1995]:

- A common design terminology improves communication.
- The use of best practices can be promoted.
- The essence of a design can be documented in a compact form.

Knowledge transfer is the unifying element in all three points. Empirical evidence shows that developers indeed use design patterns to ease communication [Hahsler, 2003]. Considering the fact that program understanding is one of the most time consuming activities of software maintenance, knowledge about applied design patterns can be useful for software maintenance.

Controlled experiments with both inexperienced [Prechtelt et al, 2002] and experienced [Prechtelt et al, 2001] software developers support the hypothesis that awareness of applied design patterns reduces:

- The time needed for software maintenance.
- The number of errors introduced during maintenance.

Because design patterns specify design constructs at a higher abstraction level than just single classes and instances, they are useful for documenting software designs [Gamma et al, 1995]. Furthermore, the choice of a specific design pattern captures the rationale behind the design and the tradeoffs that were made [Keller et al, 1999].

[Buschmann et al, 1999] classifies patterns in three categories:

- **Architectural patterns** express fundamental system organisation schemes for software systems. They specify the system-wide structural properties of an application and affect the architecture of subsystems. Architectural patterns are also called *architectural styles*. [Buschmann et al, 1999] describe several architectural styles, including layers, pipes and filters, blackboard, brokers and model-view-controller.
- **Design patterns** provide a scheme for refining the subsystems of a software system, or relationships between them. Design patterns are medium scale patterns that influence the structure of a particular subsystem, but not of the complete system. [Buschmann et al, 1999] describe several design patterns, including proxy, client-dispatcher-server and publisher-subscriber.
- **Idioms** are low-level, programming language specific patterns that describe how to implement particular aspects of components or component-relationships using the features of a given language.

3 Architecture Reconstruction

This chapter discusses several architecture reconstruction methods and tools, starting with an overview of their general properties. The selection is not exhaustive, but enumerates a representative set of methods and tools. It is based on [O'Brien et al, 2002], [Deursen, 2001], [Hassan and Holt, 2004], [Sim and Koschke, 2001] and [Bassil and Keller, 2001].

3.1 Typical scenarios

In an architecture reconstruction process (parts of) the architecture of an implemented system are recovered. This architecture is called the as-built architecture. In this process a model is extracted from the source code, after which the extracted entities are used to define higher-level abstractions. Architecture reconstruction is typically performed because uncertainty exists about the architecture of an existing system.

[O'Brien et at, 2002] describe some typical reconstruction scenarios encountered in practice:

- **View set** covers the identification of a set of architectural views that sufficiently describe a system.
- **Enforced architecture** covers the problem of consistency between the as-designed and the as-built architecture of a system.
- **Quality-attribute changes** covers the question of how changes to quality-attribute requirements affect a system. Usually it is determined how architectural patterns are used to satisfy the quality requirements and the impact of changes.
- **Common and variable artefacts** covers techniques and models for analysing the products in a domain with respect to their commonalities and differences. The aim is to find common parts in product-line systems to reduce costs.
- **Binary components** addresses the need for architecture reconstruction of systems that include COTS (binary) components. In this case only the external interfaces of the components are available (black box).
- **Mixed language** addresses the need for reconstruction methods that can analyse systems written in multiple languages or language types.

The two case studies described in this thesis are examples of the view-set, enforced architecture and mixed language scenarios.

3.2 Architecture Reconstruction Activities

[Bass et al, 2003] identify four architecture reconstruction activities; information extraction, database construction, view fusion and reconstruction. For generality we combine the first two, because many approaches do not distinguish a separate database construction activity. Furthermore, we add an additional architecture analysis activity that uses the result of the reconstruction because it sets the requirements for the preceding activities. This leads to the following four activities that can generally be distinguished in architecture reconstruction approaches:

I. **View extraction.** This activity comprises of analysing implementation artefacts such as source code and documentation, and extracting *facts* from them. In this context a fact is some piece of information about the as-built architecture that helps to understand the architecture [Ferenc et al, 2004]. Examples of facts are relations between classes (e.g. inheritance or association), information about classes (e.g. attribute and method lists), metrics (e.g. about size) and call traces.
II. **View fusion.** During this activity the extracted views are reconciled, augmented and connections are established between the elements. The aim is to improve accuracy and completeness of the view. Ideally, in the view extraction phase several complementary extractors are used, whose results are combined in the view fusion phase.
III. **Architecture reconstruction**[6]**.** The third activity creates architectural abstractions based on the fused view that collectively describe the as-built architecture. This can be done manually, possibly with tool support, or automatically. Because the reconciled views only

[6] Where the possibility of confusion exists we shall refer to the architecture reconstruction *process* and the architecture reconstruction *activity*.

represent the *results* of decisions that led to the implemented architecture, information must be added as the reconstruction proceeds. In practice human knowledge plays an important role in this process [Bass et all, 2003]. In the latter publication two sub-activities of the architecture reconstruction activity are identified; visualisation & interaction and pattern definition & recognition. The first provides mechanisms for interactive view visualisation, exploration and manipulation. The second reconstructs the architecture by detecting the code manifestations of common architectural- and design-constructs.

IV. **Architecture analysis.** During this activity the qualities of the architecture are analysed, for example to determine the conformance of the as-built architecture to the as-designed architecture, determine its scalability, or to search for reusable components. Depending on the goal, different tools may be used, varying from architecture browsers to metric extractors. Because architecture analysis is beyond the scope of this thesis it will not be discussed here further. For details the reader is referred to [Bass et all, 2003].

Some reverse engineering approaches implement all four of the above activities, but others only implement specific ones. In the remainder of this chapter approaches and tools for the architecture reconstruction process are described, starting with methods that address all four activities and followed by methods that are specialised in one specific activity.

The two case studies described in this thesis apply specific approaches to the architecture reconstruction activity. The first case study automatically detects frequently used design constructs in source code. The second automatically combines source elements into higher-level abstractions. Both implement view extraction and view fusion.

3.3 Methods & tools covering all activities

This paragraph discusses reconstruction methods and tools that cover the complete architecture reconstruction process. These methods and tools cover all four of the previously discussed architecture reconstruction activities. Subsequent paragraphs discuss methods and tools that focus on specific activities.

3.3.1 FAMOOS

FAMOOS is an acronym for Framework-based Approach for Mastering Object-Oriented Software and refers to ESPRIT project 21975. The goal of the FAMOOS project was to support the evolution of the first generation of object-oriented software with state of the art methods and tools. This is accomplished by developing a set of tools and working methods for object-oriented reengineering [Bär et al, 1999]. The developed tools are based on a common, language independent framework called Moose. Moose consists of a repository to store models for describing software systems, and facilitates access to this data.

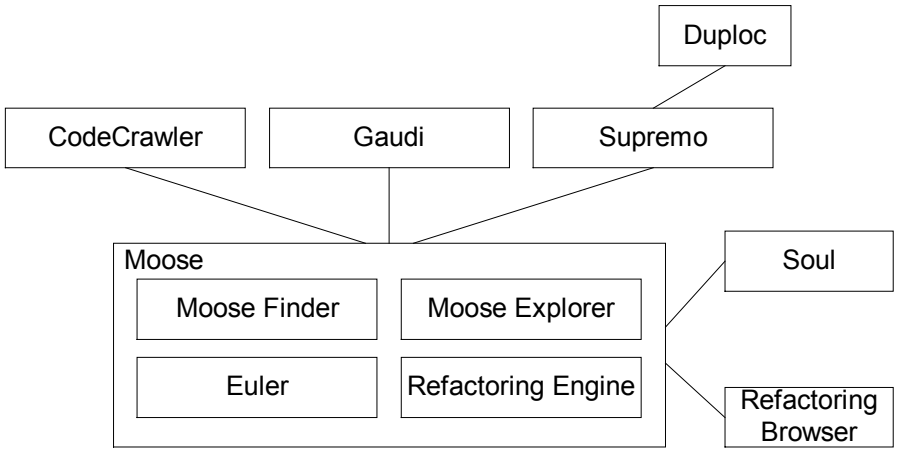


Figure 3: FAMOOS tooling (from [Ducasse, 2003])

Figure 3 shows Moose and the tooling built on top of it. The boxes represent the tools and the lines dependencies between them. The four tools that collectively form Moose are enclosed in the square labelled “Moose”.

- **Moose Finder** offers a query interface on the stored models.
- **Moose Explorer** is an application that can be used to browse through the stored models and analyse them in order to discover possible improvements.
- **Euler** is a module that computes metrics.
- The **Refactoring Engine** and **Refactoring Browser** implement language independent refactorings. These can be seen as transformations of a system’s implementation into a new form.
- **CodeCrawler** is a tool for system understanding that uses polymetric views to visualise the structure of a program [Gîrba and Lanza, 2004]. Polymetric views are based on graphs in which the nodes represent classes and the edges inter-class relationships. The node size, position and colour can be used to show up to five node characteristics. Edge width and colour can show up to two edge characteristics. The user can configure which specific characteristics are shown. Examples are the inheritance hierarchy combined with metrics like number of methods and number of different versions.
- **Gaudi** supports program understanding by incorporating dynamic information in the form of method invocations into Moose. The combination of static and dynamic information allows the creation of multiple views of an architecture.
- **Duploc** and **Supremo** implement functionality to identify and analyse duplicate code.
- **Soul** [Wuyts, 1998] implements a hybrid logic programming language in which constraints and rules about architectures can be expressed. It uses a language similar to Prolog [Fabry and Mens, 2003]. The rules and constraints can be used to check, enforce or browse architectural styles and constraints, as well as programming conventions. [Arévalo and Mens, 2002] used Soul to apply formal concept analysis to gain insight in the coupling of classes in an inheritance hierarchy. For more details on this the reader is referred to paragraph 4.1.8.

Moose uses external parsers, including Sniff+ and a Smalltalk parser. Furthermore, XMI and CDIF files can be imported from other extractors. Internally FAMOOS uses the FAMIX format (FAMoos Information eXchange Model) to store this information. Figure 4 shows the core of the FAMIX model in UML notation [Booch et al, 1999]. It consists of the main object-oriented entities, namely class, inheritance, method and attribute. Two types of relations between methods are expressed, namely invocation and accesses. An invocation represents one method calling another one, and access represents a method accessing an attribute of a class. [Demeyer et al, 1998] gives a complete description of FAMIX.

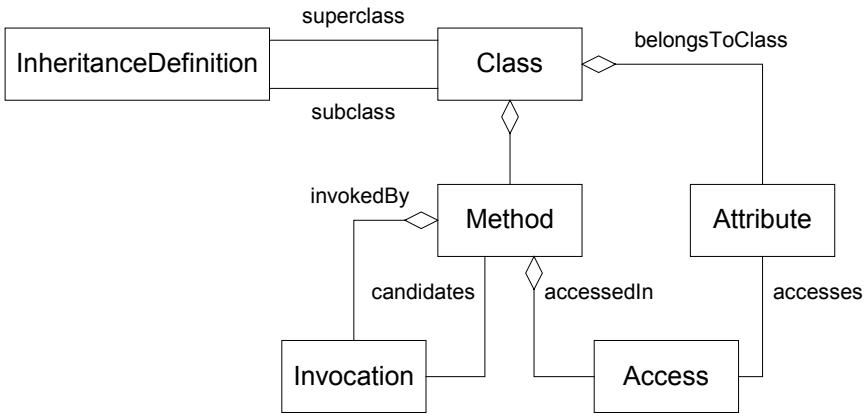


Figure 4: Core of the FAMIX model (from [Bär et al, 1999])

3.3.2 Portable Bookshelf

[Finnigan et al, 1997] introduces the concept of a "Software Bookshelf". A Software Bookshelf is a web-based system that provides easy access to large information bases that describes software systems. Distinguishing characteristics of a software bookshelf are:

- The combination of multiple, heterogeneous information sources in one hierarchically structured repository. Tools produce the raw information, which is then combined by a human librarian. Besides information extracted from source code, this includes expert knowledge and documentation like test cases, performance analysis, future plans and historical information.
- A web-based user interface that provides easy to use access with an off-the-shelf web-browser.
- An open architecture that allows integration of other reverse engineering tools. Standard, platform independent tools ensure easy integration and portability.

The Portable Bookshelf (PBS) [PBS, 2005] is a toolkit for generating a software bookshelf that implements all four activities of the architecture reconstruction process. Figure 5 shows the information flow when PBS is used to generate a bookshelf [Holt, 1997]. Fact extractors are used to extract facts from source code and export them in Rigi Standard Format (RSF). [Holt, 1997] mentions fact extractors for C, C++, Pascal and PLIX (an IBM internal language), but the generality of RSF allows easy integration of other extractors. Next, the Grok fact manipulator is used to combine the extracted facts with subsystem containment information obtained from interviews with the developers. This produces a hierarchical structure following the subsystem containment hierarchy. After the integration of the system facts automatic layout tooling is used to generate the web pages that will be shown to the user (the "shelves").

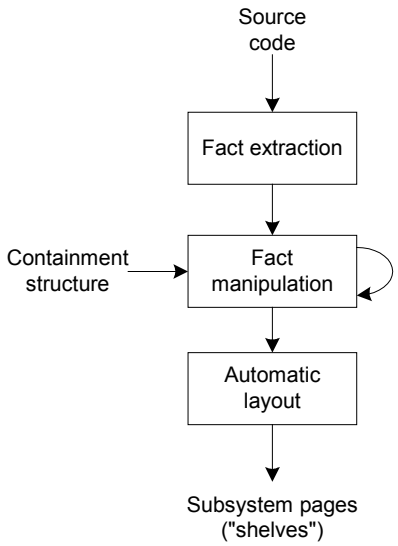


Figure 5: Process steps for bookshelf construction

PBS has been used in several case studies. [Bowman et al, 1999] describe the generation of a bookshelf for the Linux kernel. [Godfrey and Lee, 2000] describe the integration of the Acacia C and C++ fact extractors (see paragraph 3.4.2) in PBS, and its application to reconstruct the architecture of the Mozilla web browser and the VIM text editor.

[Hassan, 2002] describes an extension to PBS that supports web applications. The variety of languages often found in web applications is handled by having individual fact extractors for each language. The results of the individual extractors are combined with scripts. The combined fact information is stored in a domain model for web applications and presented to the user.

[Ivkovic and Godfrey, 2002] describe a case study where PBS is combined with the Focus method to recover the architecture of systems that use middleware technology. The combination is called Dynamo-1. Focus [Ding and Medvidovic, 2001] does not recover the complete architecture, but only the parts affected by system evolution. The approach combines static, code based recovery techniques with analysis of key use cases. The static analysis starts with extracting a source model, and packaging related classes in components. Next, an appropriate architectural style is chosen, to which the components are subsequently mapped. The use case analysis starts with important end-user functions and analyses the interaction between the architectural components during the use cases.

3.3.3 Rigi

Rigi [Rigi, 2004] supports all four activities of the architecture reconstruction process. It is developed to *"effectively represent and manipulate the building blocks of a software system and their myriad dependencies"* [Müller and Klashinsky, 1988]. This is achieved by a graph-based visualisation that supports abstraction mechanisms such as aggregation and generalisation. These mechanisms allow users to group elements and browse through the resulting structure. The grouping process is described in more detail in paragraph 6.2.2. Users can filter on arc- and node-types to display subsets of the system structure. The various operations can be automated with the Rigi Command Language.

The storage format for Rigi graphs is called Rigi Standard Format (RSF) [Wong, 1998]. Rigi contains parsers for C, C++ and COBOL that export extracted facts to this format. The RSF format is relatively well documented. Coupled with its extensibility, this led to widespread use of Rigi in reverse engineering case studies [Lanza, 2003b]. Dali and Riva for example use Rigi for visualisation, as is described in paragraph 3.3.4 and 3.3.5.

3.3.4 Riva

[Riva, 2000] describes an approach that maps the implementation artefacts to the as-designed architecture. The approach identifies six phases:

1. **Definition of architectural concepts**: in the first phase the architectural building blocks the system is composed of are determined. This can be based on the as-design architecture but also on other sources.
2. **Extraction of the source code model**: in the second phase the source code is analysed to produce a model of the source code. This may result in new architectural building blocks.
3. **Abstraction**: in the third phase the source model is mapped to the architectural building blocks found in the first phase.
4. **Improvement of architecture documents**: in the fourth phase the system's architecture is documented and its understanding is improved.
5. **Analysis of extracted architecture**: in the fifth phase an improvement plan for the architecture is produced.
6. **Architectural reorganisation of source code:** in the last phase the system is changed according to the improvement plan.

The implementation [Riva, 2000] reports uses Perl scripts to analyse C source files. The extracted information is written to RSF-files, after which Rigi is used to visualise the architecture and create abstractions.

3.3.5 Dali

Dali [Dali, 2005] is an architecture reconstruction framework in the form of a workbench that supports all four activities of the architecture reconstruction process. A *workbench* provides a lightweight framework in which other tools can easily be integrated [Bass et al, 2003]. This way support for new programming languages or visualisations can be added without affecting the existing tools or data.

Architecture reconstruction with the Dali framework comprises of four activities [Bass et al, 2003], which roughly match with the four activities identified in chapter 3:

- **Information extraction** is concerned with extracting information, mainly from source code or system traces. Dali uses a number of extraction tools, including parsers, abstract syntax tree analysers, lexical analysers, profilers and code instrumentation tools.
- **Database population** involves converting the extracted information into a standard form. Together with the information extraction activity this implements view extraction. Dali uses the PostgreSQL relational database for fact storage. Usually the extraction tools produce RSF-files, which are converted into SQL commands with Perl scripts.
- **View fusion** implements the corresponding architecture reconstruction activity. During this activity the facts the various extractors produced are combined to produce a coherent view of the architecture. View fusion is performed with SQL queries, producing a single view at an abstraction level just above that of the source code.
- **Reconstruction** is the activity where architectural abstractions are created on top of the fused view. During this activity the actual architecture is reconstructed and analysed. Dali uses Rigi for visualisation, which offers visual grouping and manipulation possibilities.

[Guo et al, 1999] describes the Architecture Reconstruction Method (ARM). ARM is a semi-automatic architecture recovery method that can be applied to systems that were developed using design patterns. [Guo et al, 1999] use the Dali workbench to perform the actual architectural recovery.
The ARM consists of four phases:

1. **Developing a concrete pattern recognition plan** for a set of design patterns. These plans are based on abstract descriptions of the patterns, which are translated into SQL queries. Design documentation and other knowledge about the system are used as starting points for this phase.
2. **Extraction of a source model** consists of extracting structural information from the source code and grouping elements into higher-level abstractions. The latter is necessary in cases where the patterns searched for are specified at a higher abstraction level than the information extracted from the source code.
3. **Detecting and evaluating pattern instances** is an automatic phase in which the pattern recognition plans are evaluated against the source model.
4. **Reconstructing and analysing the architecture** is the final phase. In this phase an analyst uses a visualisation tool such as Rigi to determine conformance of the as-built architecture to the as-designed architecture with respect to the documented design patterns. The instances of design pattern are used as indicators to form a judgment about the compliance of these two architectures.

3.3.6 Sniff+

Sniff+ [SNIFF+, 2005] is a commercial code analysis tool that implements all four activities of the architecture reconstruction process. It provides various code navigation and analysis capabilities that help developers to understand large pieces of source code. By abstracting the logical structure from the source files, Sniff+ provides an abstraction layer over the source files [Klaus, 2002]. Users can browse through the extracted symbol information. Furthermore inheritance-, component-, include- and dependency-trees can be shown.

Sniff+ supports various programming languages, including C/C++, Java, Ada 83/95, CORBA IDL and Fortran. The C/C++ compiler is designed to efficiently process large amounts of source code that can be syntactically incorrect. For a description of this compiler the interested reader is referred to [Bischofberger, 1992].

[Armstrong and Trudeau, 1998a] and [Armstrong and Trudeau, 1998b] compare the quality of various architectural extractors for software written in C, including Sniff+, Rigi, PBS, and CIA. They conclude that the parser of Sniff+ is the best of the tested parsers. It provides very few extraction errors and extracts facts with sufficient detail for architectural analysis.
[Bellay and Gall, 1997] evaluate the parsing capabilities, report generators and browsing and editing possibilities of four reverse engineering tools for C software, namely Refine/C, Imagix4D, Sniff+ and Rigi. They conclude that each tool has its strengths and weaknesses.

Strengths of Sniff+ are its fast and fault tolerant parser, graphical cross-referencer and printing capabilities. The limited graphical reporting possibilities are a weakness of Sniff+.

Because of its high quality parser, several other reverse engineering tools, including Dali and FAMOOS, use Sniff+ for fact extraction.

3.3.7 InSight

Klocwork InSight [Klocwork, 2005] is a commercial source code analysis tool that implements all four activities of the architecture reconstruction process. It can extract an architectural representation of source code written in C, C++ or Java. This information is presented in several views, both static and dynamic, which are shown in Figure 6. An arrow from view a to view b indicates b is based directly on a. Indirect dependencies between views are not shown. The straight arrows indicate automatic view construction by the fact extractor, whereas the dotted arrows indicate manual view construction with the clustering approach described in paragraph 6.2.7.

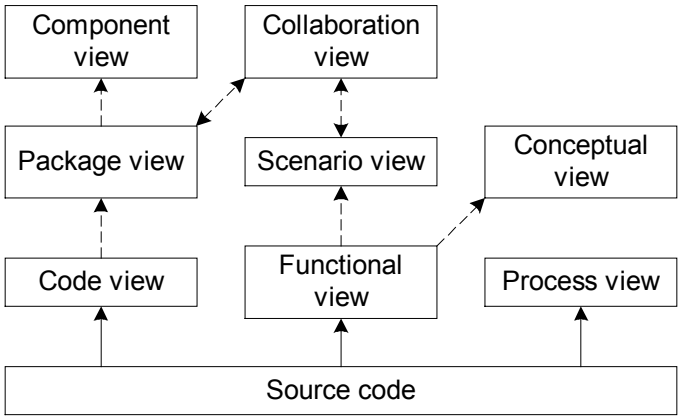


Figure 6: InSight architecture views

The views in Figure 6 each have a different purpose [Mansurov and Campara, 2003]:

- **The code view** describes how the source code, binaries and libraries are organised in the development environment. This view defines the basis of a "summary model", which is discussed in paragraph 6.2.7.
- **In the package view** elements of the code view are combined into abstract packages. This is done with three basic operations on the summary model, namely aggregation, detalization and trimming. These are also discussed in paragraph 6.2.7.
- **In the component view** packages are combined to form abstract components. This is done with the same operations as with which the package view is constructed.
- **The functional view** describes the functional relations between source code entities.
- **The scenario view** shows scenarios that are important for the architecture in terms of event sequences. Examples of event sequences are procedure calls and inter-process messages.
- **The collaboration view** projects scenarios onto component models. InSight identifies two types of collaboration views. First, collaboration diagrams integrate a structural view and a scenario by highlighting the edges that are involved in the scenario. Second, use case maps display a "time thread" through the structural view.
- **The process view** describes the system's dynamic structure in terms of processes, tasks, threads and events.
- **The conceptual view** describes the system in terms of its major architectural elements and the relations between them.

In terms of the 4+1 view model discussed in paragraph 2.4.2, InSight's conceptual view maps to the logical view in the 4+1 model. InSight's functional-, scenario- and collaboration-views map to the 4+1 scenarios and its process view maps to the 4+1 process view. InSight's code-, package- and component-views finally map to the 4+1 development view.

3.3.8 X-ray architecture recovery

X-ray is an architecture recovery approach for distributed applications [Mendonça, 1999]. It extracts the implemented executable components and their runtime interconnections. Instead of performing dynamic, runtime analysis, X-ray combines several static analysis techniques. This has the advantage that the difficulties of performing a runtime analysis are avoided [Deursen, 2001]. These include high costs, difficulties linking traces to entities in the sources, and the probe effect[7] in real-time environments.

X-ray combines three static analysis techniques:

- **Component module classification** is used to map compilation modules to executable components.
- **Syntactic pattern matching** is used to identify code fragments that implement typical component interaction features such as pipes. For each found feature this produces a set of code fragments.
- **Structural reachability analysis** is used to associate the found features to individual components. Here features are assigned to one of the involved components.

X-ray has been implemented with Prolog. Prolog facts are used to represent the information extracted from the source code, and Prolog predicates to implement the analysis techniques. The output is visualised using dotty [North and Koutsofios, 1994]. [Mendonça, 1999] reports the application of X-ray to two moderately sized distributed systems, called Samba and Field, that are both written in C. *"The results were successful in that important runtime and physical allocation aspects could be recovered"* [Mendonça, 1999].

3.4 Tools specific for view extraction activity

This paragraph discusses several tools that primarily cover the view extraction activity of the architecture reconstruction process. Tools such as Sniff+ and InSight, that cover all architecture reconstruction activities, including view extraction, have already been discussed in the previous paragraph.

3.4.1 Columbus/CAN

Columbus/CAN is a reverse engineering framework for view extraction from C++ code [Ferenc et al, 2004]. It provides a general framework that combines a number of reverse engineering tasks, but mainly focuses on the view extraction activity. Columbus provides a common interface for plug-ins, some of which are shipped with the framework itself. With these plug-ins Columbus supports project handling, data extraction, data representation, data storage, filtering and visualisation [Ferenc et al, 2001]. Other plug-ins can be developed with the plug-in API.

Three types of Columbus plug-ins can be distinguished:

- **Extractor plug-ins** analyse a given input file and produce an output file that contains the extracted facts in Columbus' internal representation. Columbus is shipped with an extractor for C++ called CAN (C++ ANalyser) [Columbus, 2003]. CAN has an embedded C++ processor, but can also wrap other C++ compilers for support of proprietary constructs.
- **Linker plug-ins** build and merge the internal representations of the project. For more information on the schema Columbus uses for this representation the interested reader is referred to [Columbus, 2003].
- **Exporter plug-ins** convert the internal representation built by the linker to a specific output format. Columbus is shipped with exporter plug-ins for various formats, including GXL, UML XMI, Famix XMI and RSF [Ferenc et al, 2004]. The multitude of export formats is an important strength of Columbus [Kersemakers, 2005].

[7] The probe effect is the effect that by implementing points of observation in the software the timing-related behaviour of the software is influenced.

Columbus has been used in many reverse engineering projects. For example Alborz, Sart and Maisa all[8] use Columbus/CAN to extract facts from source code to detect design pattern instances in the as-built architecture.

3.4.2 Acacia

[Chen et al, 1998] describe a C++ data model for reachability analysis and dead code detection. The Acacia system implements this model. Acacia uses the CCIA fact extractors to process C and C++ code, and the older CIA fact extractor for C code. A simple database is used to store the extracted facts, which are manipulated with a flat-file query language. Finally, Dot [Graphviz, 2005] is used to visualise the results.

Acacia supports all four activities of the architecture reconstruction process, but in literature the fact extractors are most commonly referred to. These are used in several reverse engineering approaches, including PBS and Bunch, which are described in paragraph 3.3.2 and 6.2.4 respectively.

3.5 Approaches specific for architecture reconstruction activity

Earlier in this chapter tools and methods implementing all architecture reconstruction activities have been discussed. All these support the architecture reconstruction activity. However, many specific tools and methods for the architecture reconstruction activity exist. This paragraph discusses a non-exhaustive selection of several approaches that are relevant to the case studies described in this thesis.

3.5.1 Manual approaches

Manual architecture reconstruction approaches use navigation and browsing tools to manually reconstruct an architecture. Shrimp is an example of such a tool, but many others exist. Shrimp is described here because it is used in the case studies described later in this thesis.

Shrimp (Simple Hierarchical Multi-Perspective) is a visualisation and navigation tool for large, hierarchical, information spaces. In the context of architecture reconstruction it can be used to visualise and navigate through the architecture. Shrimp's primary view is a zoom interface that combines the hypertext-browsing metaphor with animated zooming over nested graphs [Storey et at, 2001]. The hierarchical structure is visualised through a nested graph with the parent-child relationship visualising subsystem containment. Additional relationships are visualised with coloured arcs over the nested graph.

Shrimp can show subsystems in graphical and textual views, which can be divided in four categories [Michaud et al, 2001]:

- Source code artefacts and relationships.
- Architectural abstractions and relationships.
- Documentation.
- Metrics and other analysis results.

[Bassil and Keller, 2001] describe the results of a survey on software visualisation tools and the functionality in practice desired from these tools. Several of these tools are mentioned, including Shrimp, Sniff+, Rigi, Fujaba and PBS. Because it is beyond the scope of this thesis, the other visualisation tools are not discussed here. For more information on those the reader is referred to [Bassil and Keller, 2001].

3.5.2 Pattern detection based architecture reconstruction approaches

Pattern-based architecture reconstruction approaches detect instances of common constructs, or patterns, in the implementation. By replacing these instances with an abstracted form a simplified view of the architecture is created. During reverse engineering,

[8] These tools are described in paragraph 4.1.7, 4.1.9 and 4.1.4 respectively.

engineers quickly recognise these abstractions, which reduces the time needed for program understanding. Pattern-based reconstruction approaches are often based on structural information, thus searching for structural design patterns. Alternatively, behavioural patterns or repeated code fragments can be searched for.

In practice pattern detection based approaches do not cover all entities in the software [Quilici, 1995]. The reason for this is that in practice software is never completely composed of repeated structures. So these approaches produce architectural views in which a subset of the program entities is converted to an abstract form, while the other entities remain at an abstraction level just above that of the code.

According to [Sartipi and Kontogiannis, 2003], pattern detection based approaches are one of the prominent methods for automatic architecture reconstruction (together with clustering-based approaches). This thesis describes a case study where instances of structural design patterns are detected in industrial software. Related work concerning pattern-based architecture reconstruction is discussed in more detail in chapter 4.

3.5.3 Using clustering techniques for architecture reconstruction

Clustering-based architecture reconstruction techniques use clustering techniques to find architectural components in source code. Clustering techniques find some natural ordering of data elements, in this case source code elements.

The algorithms usually start with an abstract graph that represents the structure of the program, for example with the nodes representing classes and the edges inter-class relationships. Some similarity measure is then used to find groups of classes that belong together, which are grouped into subsystems. This is repeated until an optimal decomposition is found. The end result represents an architectural view with abstract entities that group multiple source code entities.

The similarity measure determines the properties of the produced decomposition. Typically, similarity measures attempt to achieve high cohesion within modules and low coupling between modules. This is based the criteria used for the decomposition of software systems for which [Parnas, 1972] and [Parnas et al, 1984] laid the foundations. [Booch, 1994] extends this to object-oriented software, stating that systems should be composed of collaborating agents (objects). To simplify their understanding, objects should be organised into hierarchies that promote strong cohesion and loose coupling. [Sommerville, 2004] confirms that well designed systems exhibit high cohesion and low coupling.

When reconstructing architectures with clustering techniques a ‘natural’ structure of the software is determined. There is a difference however, between discovering an architecture and imposing one. Clustering techniques impose a new ordering, instead of discovering some hidden ordering [Wiggerts, 1997]. This ordering must be evaluated on its usefulness for program comprehension [Tzerpos and Holt, 1998].

According to [Sartipi and Kontogiannis, 2003], clustering-based approaches are one of the two prominent methods for automatic architecture reconstruction (together with pattern detection based approaches). This thesis describes a case study where the architecture of an industrial system is reconstructed with clustering techniques. Related work concerning clustering-based architecture reconstruction is discussed in more detail in chapter 6.

3.5.4 Using Conway's law for architecture reconstruction

Besides technical factors such as the functional and non-functional requirements, organisational factors also play an important role during the development of software systems. [Conway, 1968] states that *"organisations which design systems are constrained to produce designs which are copies of the communication structures of these organisations"*. This has become known as Conway's law [Brooks, 1995], [Demeyer et al, 2004] [Herbsleb and Grinter, 1999]. Conway’s law can be used during the architecture reconstruction activity to choose suitable abstractions.

[Bowman and Holt, 1998] describe experiments where the ownership architecture is compared to the as-built and as-designed architecture. These experiments used three large software systems as input; Linux (800 KLOC of C code), Mozilla (1500 KLOC of C and C++ code) and Aleph, a commercial software development system (3500 KLOC of C and C++ code). The experiments showed that the ownership architecture predicts the as-built architecture very well, and is closely correlated with the as-designed architecture. In fact, the ownership architecture predicts subsystem dependencies in the as-built architecture at least as good as the as-designed architecture. The as-designed architecture tends to underestimate dependencies in the as-built architecture. The ownership architecture on the other hand tends to overestimate them. This makes it difficult to perform architecture reconstruction solely based on the ownership architecture, but in conjunction with the as-designed architecture it provides a good starting point.

3.5.5 Slicing based architecture reconstruction approaches

Program slicing is a technique that extracts those program elements from the source code that affect the behaviour of the program at a certain point. In this context a point is for example a certain line of the code, or an interface. Together the extracted elements form a specific view of the program that is called a program slice. A *program slice* consists of those parts of the program that affect the behaviour of the program at the chosen point, either directly or indirectly.

According to [Beck and Eichmann, 1993] slices are usually generated by first building a program representation that contains a program dependence chart. This is done using data- and control-flow analysis. In this graph the nodes represent the entities in the program and the edges dependencies. Usually program statements are chosen as entities. A disadvantage of this choice is that these entities have a relatively low abstraction level. [Beck and Eichmann, 1993] use interface entities such as procedures and global variables to achieve a slightly higher abstraction level.
After the program dependence graph is constructed the generation of the slices is straightforward. Starting with the entities of interest, which are specified in the slicing criterion, the edges in the graph are followed to generate their transitive closure.

In the context of reverse engineering slicing is often used to extract reusable components from an implementation or specification. [Beck and Eichmann, 1993] for example use slicing to detect which parts of a component affect some subset of its interface.

Slicing is a very useful technique for reverse engineering. But because it is not related to the two methods applied in the case studies described in this thesis, we shall not discuss it in more detail. For more information the interested reader is referred to [Beck and Eichmann, 1993] and [Zhao, 2000].

4 Pattern detection in source code

One of the two case studies described in this thesis aims to detect design pattern instances in source code. This chapter discusses similar approaches reported in literature. The discussed approaches are divided in two categories; approaches in the first category need upfront knowledge of the expected patterns, whereas those in the second do not have this limitation. The approach used in the case study falls in the second category.

4.1 Detecting known patterns

This paragraph describes approaches that detect instances of *known* patterns in source code. These approaches generally use a *pattern library*. In this library the patterns are specified in some specification language. These specifications are matched against a model extracted from the design documentation or source code to find the patterns.
Although not exhaustive, this paragraph gives a representative overview of these approaches. It is based on references found in literature, using [Kersemakers, 2005] as a starting point.

4.1.1 Pat

[Krämer and Prechtelt, 1996] describe the Pat system, which treats pattern detection as a constraint satisfaction problem. In Pat patterns are described with a set of propositions. The fundamental idea is to check which propositions hold for a model of the system. More precisely; let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of predicates modelling the system under investigation ($s_i \equiv true,\ 1 \le i \le n$) and $P = \{p_1, p_2, \ldots, p_m\}$ a set of propositions expressing design patterns. Pattern p_j ($1 \le j \le m$) is present in the software if and only if p_j can be inferred from S. The Prolog Engine performs this inference. The instantiated values for the variables in p_j specify the entities involved in the pattern expressed by p_j.

Figure 7 illustrates how this process is implemented in Pat. The Code Analysis module extracts S from C++ header files and converts it to Prolog facts automatically. The Pattern-to-Prolog module accepts a set of patterns as input and produces the Prolog rules that express P. This is done in two steps; first the patterns are expressed as static OMT diagrams [Rumbaugh, 1990] manually, after which they are translated to Prolog rules automatically. The Prolog Engine uses the output of these two modules to calculate the set of pattern candidates C.

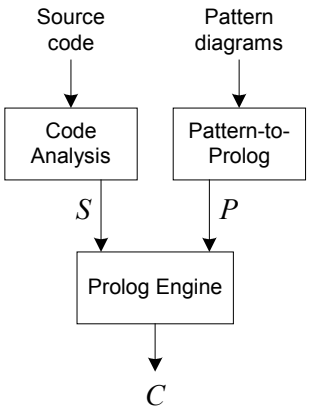


Figure 7: Dataflow in Pat

[Krämer and Prechtelt, 1996] expressed five structural design patterns as Prolog rules, namely Adapter, Bridge, Composite, Decorator and Proxy [Gamma et al, 1995]. An important limitation of the method is that behavioural patterns are difficult to detect. This is caused by the fact that their distinguishing characteristics are difficult to extract from source code alone, or are simply not extracted by the chosen extractor. For similar reasons some structural design patterns cannot be detected reliably either; there is not enough semantic information

available to achieve this. These limitations lead to false positives in the list of pattern candidates. [Krämer and Prechtelt, 1996] applied Pat to four C++ programs, consisting of 9 to 343 classes. They report a precision between 14 and 50 percent and 100 percent recall. The false positives in the result must be sorted out manually.

4.1.2 AOL Graphs

[Antoniol et al, 1998] describe a method for to find design patterns that uses metrics and delegation constraints to filter out false positives. It takes both code and design documents as input, as is shown in Figure 8. Both are translated to an intermediate representation called AOL (Abstract Object Language). The Pattern Recogniser uses these together with the AOL pattern descriptions from the AOL Pattern Library as input. After parsing the AOL graph it calculates various metrics from it, for example the number of methods and operations of a class, or the number of associations a class is involved in. The Constraint Evaluator then executes three matching operations:

1. The metrics are used to filter out classes that cannot be involved in the searched patterns. This is done before matching the AOL pattern specifications to reduce the search space for operation 2.
2. The AOL pattern specifications are matched against the input graphs.
3. Delegation constraints are evaluated against the pattern candidates. A delegation constraint specifies call delegation behaviour for classes.

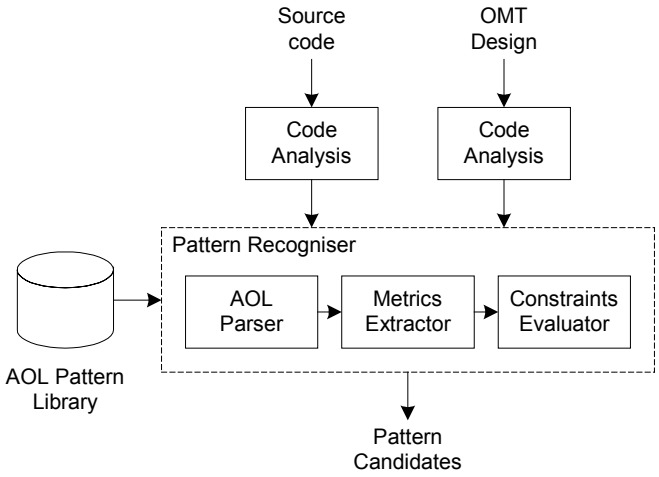


Figure 8: Pattern detection process

[Antoniol et al, 1998] describe a Java implementation of the described method that searches for five structural design patterns, namely Adapter, Bridge, Proxy, Decorator and Composite [Gamma et al, 1995]. This application has been applied to fourteen programs written in C++, including both public domain and industrial code. The programs varied in size between 4 and 50 KLOC. On average the precision of the result is about 55 percent and recall 100 percent. The use of the metrics and checking of delegation constraints reduced the number of false positives indeed. This conclusion is mainly based on the experience with the public domain code. In the industrial code only a few pattern instances are found.

In some cases the authors found differences between the patterns detected in the design documentation and in the source code. This is attributed to deviations between the as-designed and the as-built architecture. In case of the industrial code the sets of patterns detected from the design and from the code are disjoint, indicating large differences between these two architectures.

4.1.3 Spool

[Keller et al, 1999] are the first to explicitly link design pattern detection to reverse engineering. They state that design patterns capture the rationale and trade-offs in a software design. This way knowledge about applied design patterns helps to understand a design.

An important problem when searching for design patterns in existing software is that *"patterns can be implemented in many different ways"* [Keller et al, 1999]. This is handled by utilising the cognitive strength of a human analyser.

The presented environment, called Spool, supports automatic, semi-automatic and manual searching. It stores facts extracted from source code in the Poet object-oriented database, using a schema based on the UML meta-model. Design pattern candidates are presented in a graphical user interface with which a human analyst can check them. The layouts in this user interface are generated with Dot and Neato [Graphviz, 2005]. The detection of three design patterns is implemented, namely Template Method, Factory Method and Bridge [Gamma et al, 1995]. The analysis of three C++ programs ranging in size between 70 and 472 KLOC is described. Although the detection results are described, no figures are presented for precision and recall. This is done because the strength of the Spool environment is not so much the quality of the automatic detection, but the integration of the human in the recovery process. [Keller et al, 1999] state that this is the only way to find interesting patterns.

4.1.4 Maisa

Maisa is a tool for the analysis of software architectures that is based on the detection of design pattern instances. It predicts the quality of designs by searching design-level UML diagrams [Booch et al, 1999] for a set of design patterns and anti-patterns that are expressed as Prolog facts [Paakki et al, 2000]. The tool matches these with the facts extracted from UML designs. The resulting set of patterns is then used to extract quality-relevant metrics. Besides these metrics, conventional metrics about the entire architecture (e.g. number of classes) are also used. A human analyst can use the produced metrics, together with the properties of the patterns, to determine the quality of the architecture.

[Paakki et al, 2000] have applied Maisa to a large telecommunications system. The approach was able to detect design patterns efficiently and reliably and was able to find several known design problems.

[Ferenc et al, 2001] describes the integration of Columbus/CAN and Maisa. Columbus is used to extract a set of UML class diagrams from C++ code. These are then imported into Maisa and analysed. For more information on Columbus/CAN the reader is referred to paragraph 3.4.1. The multitude of implementation variants is handled with *partial satisfyability*, which means that the Prolog inference engine also accepts rules that do not match the propositions completely.

[Ferenc et al, 2001] report that a lack of information in the design diagrams caused problems for the pattern identification. Possible solutions are to search for the patterns using less information or to use partial satisfyability. However, both would increase the number of false positives. The lack of information also made it impossible to distinguish patterns with a similar structure, but different behaviour, such as Bridge and Command [Gamma et al, 1995].

4.1.5 Idea

[Bergenti and Poggi, 2000] describe the Idea tool (Interactive DEsign Assistant), an interactive design assistant that automatically searches for design pattern instances and produces critiques about their implementation. Idea takes an UML design exported in XMI format as input and detects instances of the following design patterns [Gamma et al, 1995]:

Proxy	Factory Method
Adapter	Abstract Factory
Bridge	Iterator
Composite	Observer
Decorator	Prototype

These patterns are expressed as Prolog rules, which are matched with facts extracted from the input file. Information extracted from collaboration diagrams is used to check if a detected pattern instance has the required object interactions. If this is the case, the instance is shown to the user. For each pattern a set of design rules is stored in the knowledge base, with the

corresponding critiques. If a certain rule is violated the corresponding critique is fired and presented to the user.

4.1.6 Fujaba

Fujaba (From UML to Java And Back Again) is a public domain research prototype of a CASE tool [Klein et al, 1999]. Fujaba supports both forward- and reverse engineering for UML class- and behaviour-diagrams. This means that Java code can be generated from these diagrams, and that these diagrams can be generated from source code.

[Niere et al, 2001] and [Niere et al, 2003] describe a design pattern recognition method for the Fujaba environment. Like other methods, this method uses a pattern library. The problem of having many design variants is handled by composing the stored patterns of smaller sub-patterns. These capture the generic parts of the pattern. The individual design pattern variants are composed of these sub-patterns. This reduces the total number of different patterns that must be detected. [Niere et al, 2001] describe an implementation that detects the Composite pattern [Gamma et al, 1995].
Implementation differences are handled with fuzzy reasoning. The sub-patterns are specified by a set of detection rules, each of which is associated with a *fuzzy belief* f ($0 \leq f \leq 1$). During the detection phase a design pattern is only detected if the accumulated fuzzy beliefs of the matching rules exceed a user-specified threshold. Besides detection rules, contra indications can also be specified to improve the detection process. The actual detection is implemented with Generic Fuzzy Reasoning Nets [Jahnke et al, 1997], which are expanded into Fuzzy Petri Nets. The detection rules are implemented as graph rewrite rules.

The above pattern detection cannot check if the reported instances implement the dynamic behaviour the design pattern prescribes. This causes false positives, especially if behavioural patterns are searched for. [Wendehals, 2003] and [Heuzeroth et al, 2002] independently report extensions to Fujaba that use dynamic analysis to reduce the number of false positives. In both cases the dynamic analysis is applied to the output of the static analysis.

[Heuzeroth et al, 2002] consider a design pattern's protocol as a set of state transitions. When a node n is detected to be involved in a state transition it is checked whether or not n is part of any pattern instance the static analysis reported. The pattern library is extended with a set of specific rules that specify the protocol of each pattern. If n is part of a pattern instance the appropriate rules are checked.
When the analysis is complete pattern candidates are partitioned into four groups:

- **Full match**: all rules in the protocol of the pattern candidate are completely executed.
- **May match**: at least one, but not all of the rules of the protocol are executed.
- **Mismatch**: the protocol requirements are violated.
- **No decision**: none of the monitored nodes of the candidate are executed.

The dynamic information is gathered using an on-line debugger and automatic code instrumentation. The first method causes severe performance problems, which makes it impossible to use in practice. The second has the disadvantage of requiring an extra compilation, but this is considered acceptable. Note that both methods are vulnerable to the choice of the execution scenario. A pattern that is not executed cannot be detected.
[Heuzeroth et al, 2002] reports the results of an experiment in which the Observer pattern is searched for in two medium sized Java programs. [Heuzeroth et al, 2003] report a case study where the Observer, Composite, Mediator, Chain of Responsibility and Visitor patterns are searched. In both cases the addition of the dynamic analysis removed almost all false positives.
A disadvantage of the previously described method is the need to specify the detection rules manually. [Heuzeroth et al, 2003] describe a specification language in which the constraints that define a design pattern can be expressed. The part of the language expressing the static constraints is based on predicate calculus. The part expressing the dynamic constraints defines pre- and post-conditions. From such a specification, the detection rules are generated automatically.

[Wendehals, 2003] describes a different method to reduce the number of false positives with dynamic information. The proposed method expresses a design pattern's protocol as graph rewrite rules, similar to the static analysis. These rules are extracted from message sequence diagrams. The dynamic information is retrieved from method traces.

4.1.7 Alborz

Alborz is a prototype toolkit for recovering the architecture of systems written in procedural languages [Sartipi, 2001]. The tool provides two techniques for architecture recovery; pattern recognition and clustering. In this paragraph the pattern detection is described, whereas paragraph 6.2.5 describes the clustering method.

The pattern recognition of Alborz represents the analysed software with an attributed relational graph in which the nodes represent files, functions, datatypes and variables [Sartipi and Kontogiannis, 2003]. The edges represent "call" and "use" relationships. The patterns are expressed as AQL queries (Architectural Query Language). These are translated to relational graphs too, after which a graph-searching algorithm is used to find pattern instances. The found instances are visualised with Rigi.

4.1.8 Using Soul

[Fabry and Mens, 2003] describe the use of the Soul language to detect patterns in Java and Smalltalk code. Soul is a programming language for logic reasoning, similar to Prolog. Like Prolog, Soul has a logical inference engine. The method is similar to that of Pat, but uses more information. Besides header information, the proposed method also takes method bodies into account. This includes method invocations and variable accesses.

[Fabry and Mens, 2003] applied the described method to two applications written in Java and Smalltalk (38 to 377 classes). In the reported case study the Double Dispatch and Getting Method patterns [Beck, 1997] are searched for. The results were validated manually, which revealed neither false positives nor false negatives.

4.1.9 Sart

[Kersemakers, 2005] describes a case study where design pattern are detected in a way similar to the method used in Pat. This method is implemented in the Sart tool (Software Architecture Recovery Tool). Design patterns are expressed as Prolog rules and the facts that are extracted from the source code as Prolog facts. Detection rules were implemented for the Observer, Interceptor, Pipe-and-Filter and Blackboard patterns [Gamma et al, 1995], [Buschmann et al, 1999].

To reduce the number of false positives, behavioural information is extracted from the code. More specific, the set of methods S that can be called from the scope of each method m is extracted. S is called the *reach* of m. Each design pattern has a specific sequence of method calls. The reach is used to determine if a potential pattern instance can implement the method-calling sequence the pattern prescribes. [Kersemakers, 2005] reports that adding the reach-based filtering significantly reduced the number of false positives.

The multitude of pattern implementation variants is handled by introducing a set of relaxation strategies for each pattern. These represent the variants typically used in practice.

[Kersemakers, 2005] uses Columbus/CAN for fact extraction. Difficulties are reported with the extraction of associations based on attributes, which made it more difficult to reduce the number of false positives. Rigi is used to visualise pattern instances.

4.1.10 Backdoor

[Shull et al, 1996] present an inductive method to help discover custom, domain specific design patterns in existing object-oriented code. The proposed method uses a knowledge base of existing patterns and a six-step process to check if a suspected pattern instance really matches an instance in the knowledge base [Shull et al, 1996]:

1. Review the problem specification and design documents. This provides insight in the problem at hand, and incorporates existing knowledge.
2. Using the class declarations, develop a preliminary model of the system.
3. Refine the preliminary model with information from class implementations.
4. Identify candidate patterns in the refined model based on inheritance and communication links between classes. In this step the actual pattern detection takes place.
5. Analyse the detected pattern candidates to find useful design patterns. This is the most labour-intensive step and requires a skilled analyst.
6. Interview designers and implementers to check the suspected architecture and obtain information about the rationale.

4.2 Detecting unknown patterns with FCA

The use of a pattern library requires upfront knowledge about the implemented patterns. This paragraph describes the use of a mathematical technique called Formal Concept Analysis (FCA) to detect various types of patterns. The selection of methods and tools is based on [Snelting, 2000], [Snelting] and [Tilley et al, 2003]. Because one of the case studies described later in this thesis uses FCA for pattern detection, the approaches in this chapter are described in more detail than the earlier ones.

4.2.1 Formal Concept Analysis introduction

Formal Concept Analysis (FCA) is a mathematical technique to identify "*sensible groupings of formal objects*[9] *that have common formal attributes"* ([Siff and Reps, 1998] citing [Wille, 1981]). FCA is also known as Galois lattices ([Arévalo et al, 2003] citing [Wille, 1981]). The analysis starts with a *formal context*, which is a triple $C=(O,A,R)$ in which O is the finite set of formal objects and A the finite set of formal attributes. R is a binary relation between elements in O and A, hence $R \subseteq O \times A$. If $(o,a) \in R$ it is said that object o *has* attribute a.

Let $X \subseteq O$ and $Y \subseteq A$. Then the *common attributes* $\sigma(X)$ of X and *common objects* $\tau(Y)$ of Y are defined as [Ganter and Wille, 1998]:

$$\sigma(X) = \left\{ a \in A \middle| \forall o \in X : (o,a) \in R \right\} \tag{1}$$

$$\tau(Y) = \left\{ o \in O \middle| \forall a \in Y : (o,a) \in R \right\} \tag{2}$$

The following derivation operators hold for any $X, X_1, X_2 \subseteq O$ and $Y, Y_1, Y_2 \subseteq A$ [Ganter and Wille, 1998]:

$$\begin{aligned} X_1 \subseteq X_2 &\Rightarrow \sigma(X_2) \subseteq \sigma(X_1) \\ Y_1 \subseteq Y_2 &\Rightarrow \tau(Y_2) \subseteq \tau(Y_1) \end{aligned} \tag{3}$$

$$\begin{aligned} X &\subseteq \tau(\sigma(X)) \text{ and } \sigma(X) = \sigma(\tau(\sigma(X))) \\ Y &\subseteq \sigma(\tau(Y)) \text{ and } \tau(Y) = \tau(\sigma(\tau(Y))) \\ X &\subseteq \tau(Y) \Leftrightarrow Y \subseteq \sigma(X) \end{aligned} \tag{4}$$

A *formal concept* of the context (O,A,R) is a pair of sets (X,Y), with $X \subseteq O$ and $Y \subseteq A$, such that [Ganter and Wille, 1998]:

$$Y = \sigma(X) \wedge X = \tau(Y) \tag{5}$$

[9] Be aware that formal objects and formal attributes are not the same as objects and attributes in object-oriented programming.

Informally a formal concept is a *maximal* collection of objects sharing common attributes. X is called the *extent* and Y the *intent* of the concept.

For example consider the following six sports: swimming, soccer, waterpolo, icehockey, triathlon and bobsledding. These sports are characterised with five properties: the fastest wins, water/ice involved, players running on foot, ball used and teamsport or individual sport. Suppose the sports must be organised according to their properties. Then in FCA terms the sports are the formal objects and their properties the formal attributes. Table 3 shows the relation between the objects and the attributes. For example soccer is a teamsport where the players run on foot and a ball is used. Further, the only teamsports where a ball is used are soccer and waterpolo.

		Formal Attributes				
		Fastest	Water	Running	Ball	Team
Formal Objects	Swimming	√	√			
	Soccer			√	√	√
	Waterpolo		√		√	√
	Icehockey		√			√
	Triathlon	√	√	√		
	Bobsledding	√	√	√		√

Table 3: A characterisation of sports

In the example ({soccer, waterpolo},{ball, team}) is an example of a concept, but ({triathlon, bobsledding},{fastest, water}) is not because swimming also has these attributes. ({soccer, waterpolo},{running, ball, team}) is not a valid concept either because waterpolo does not have the running attribute.

The extents and intents can be used to relate formal concepts hierarchically. For two formal concepts (X_0, Y_0) and (X_1, Y_1) [Ganter and Wille, 1998] define the subconcept relation $\leq$ as:

$$(X_0, Y_0) \leq (X_1, Y_1) \Leftrightarrow X_0 \subseteq X_1 \Leftrightarrow Y_1 \subseteq Y_0 \tag{6}$$

If p and q are formal concepts and $p \leq q$ then p is said to be a *subconcept* of q and q is a *superconcept* of p. For example ({soccer}, {running, ball, team}) is a subconcept of ({soccer, waterpolo},{ball, team}). The subconcept relation enforces an ordering over the set of concepts that is captured by the supremum $\coprod$ and infimum $\prod$ relationships. They define the *concept lattice* L of a formal concept C with a set of concepts I [Ganter and Wille, 1998]:

$$\coprod_{(X_i,Y_i)\in I} (X_i, Y_i) = \left(\tau\left(\sigma\left(\bigcup_{(X_i,Y_i)\in I} X_i \right) \right), \bigcap_{(X_i,Y_i)\in I} Y_i \right) \tag{7}$$

$$\prod_{(X_i,Y_i)\in I} (X_i, Y_i) = \left(\bigcap_{(X_i,Y_i)\in I} X_i, \sigma\left(\tau\left(\bigcup_{(X_i,Y_i)\in I} Y_i \right) \right) \right) \tag{8}$$

where I is the set of concepts to relate. To calculate the supremum $\coprod$ (or smallest common superconcept) of a set of concepts their intents must be intersected and their extents joined. The latter set must then be enlarged to fit to the attribute set of the supremum. The infimum $\prod$ (or greatest common subconcept) is calculated in a similar way.
For example the supremum c_7 of c_1=({soccer},{running, ball, team}) and c_2=({waterpolo},{water, ball, team}) is calculated as follows:

$$c_7 = c_1 \coprod c_2$$
$$c_7 = \left(\tau\left(\sigma\left(\{soccer\} \cup \{waterpolo\} \right) \right), \{running, ball, team\} \cap \{water, ball, team\} \right)$$
$$c_7 = \left(\tau\left(\{ball, team\} \right), \{ball, team\} \right)$$
$$c_7 = \left(\{soccer, waterpolo\}, \{ball, team\} \right)$$

[Siff and Reps, 1997] describe a simple bottom-up algorithm that constructs a concept lattice L from a formal context $C=(O,A,R)$ using the supremum relation. It starts with the concept with the smallest extent, and constructs the lattice from that concept onwards. The algorithm utilises that for any concept (X,Y) [Snelting, 1996]:

$$Y = \sigma(X) = \sigma\left(\bigcup_{o \in X} \{o\}\right) = \bigcap_{o \in X} \sigma(\{o\}) \tag{9}$$

This equation enables calculating the supremum of two concepts by intersecting their intents. (10) gives a formalised description of the lattice construction algorithm. This description is based on the informal description by [Siff and Reps, 1997].
Stated informally, the algorithm starts with the calculation of the smallest concept c_b of the lattice. The set of atomic concepts, together with c_b, is used to initialise L. Next the algorithm initialises a working-set W with all pairs of concepts in L that are not subconcepts of each other. A hash table is used to store L and allow efficient checking for duplicates later on. The algorithm subsequently iterates over W to build the lattice using the supremum relation for each relevant concept-pair. The supremum of two concepts is calculated using (9). Recall that in this calculation the intents of the concepts c_1 and c_2 are intersected, after which τ is applied obtain the extent. If the calculated concept is new it is added to L and the working-set is extended with relevant new concept pairs.

$$\begin{aligned}
&c_b := \left(\tau(\sigma(\varnothing)), \sigma(\varnothing)\right)\\
&L := \{c_b\} \cup \left\{\left(\tau(\sigma(o)), \sigma(o)\right) \mid o \in O\right\}\\
&W := \left\{(c_1, c_2) \in L^2 \,\middle|\, \neg(c_1 \le c_2 \vee c_2 \le c_1)\right\}\\
&\text{for each } (c_1, c_2) \in W \text{ do}\\
&\qquad c' = c_1 \sqcup c_2\\
&\qquad \text{if } c' \notin L \text{ do}\\
&\qquad\qquad L := L \cup \{c'\}\\
&\qquad\qquad W := W \cup \left\{(c, c') \,\middle|\, c \in L \wedge \neg(c \le c' \vee c' \le c)\right\}\\
&\qquad \text{od}\\
&\text{od}
\end{aligned} \tag{10}$$

Figure 9 and Table 4 show the result of applying algorithm (10) to the sports example. c_b and c_t are the bottom and top concepts respectively.

c_t	({swimming, soccer, waterpolo, icehockey, triathlon, bobsledding}, ϕ)
c_0	({swimming, triathlon, bobsledding}, {fastest, water})
c_1	({soccer}, {running, ball, team})
c_2	({waterpolo}, {water, ball, team})
c_3	({icehockey, waterpolo, bobsledding}, {water, team})
c_4	({triathlon, bobsledding}, {fastest, water, running})
c_5	({bobsledding}, {fastest, water, running, team})
c_6	({swimming, triathlon, bobsledding, icehockey, waterpolo}, {water})
c_7	({soccer, waterpolo}, {ball, team})
c_8	({soccer, waterpolo, icehockey, bobsledding}, {team})
c_9	({soccer, triathlon, bobsledding}, {running})
c_{10}	({soccer, bobsledding}, {running, team})
c_b	(ϕ, {fastest, water, running, ball, team})

Table 4: Extents and intents of the sports example

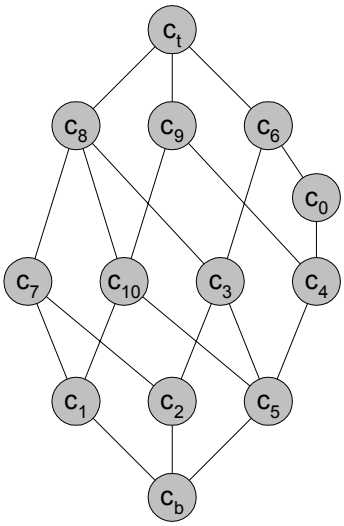


Figure 9: Concept lattice for sports example

The time complexity of algorithm (10) depends on the number of lattice elements. If the context contains n formal objects and n formal attributes, the lattice contains 2^n concepts [Snelting, 1996]. This means the worst case running time of the algorithm is exponential in n. In practice however, the size of the concept lattice typically[10] is $O(n^2)$, or even $O(n)$. This results in a typical running time for the algorithm of $O(n^3)$ [Snelting, 1996].

Algorithm (10) is a very simple lattice construction algorithm that does not perform very well. [Ganter, 1987] presents the "NextConcept" algorithm. The computational complexity of this algorithm is linear with the size of the concept lattice, but more difficult to understand. A disadvantage of the NextConcept algorithm is that it only produces the concepts, and not the relations between them. For cases where the lattice is needed, [Lindig, 2002] presents the "Lattice" algorithm. This algorithm produces both the concepts and the relations between them. Worst case it has a quadratic complexity, but since the quadratic component is relatively small, the algorithm's time complexity can be considered $O(n)$. For more lattice construction algorithms the interested reader is referred to [Kuznetsov and Obëdkov, 2001].

4.2.2 Early uses of FCA for reverse engineering

[Snelting, 1996] first described the use of FCA in the context of reverse engineering. He describes the application of concept analysis to the problem of reengineering compile-time configurations in procedural code. These configurations are defined by means of preprocessing instructions (#if…#endif) in the source files. The formal context $C=(O,A,R)$ is used with:

- O: set of source-code fragments.
- A: set of used preprocessor symbols
- R: usage of the preprocessor symbols in the source-code fragment.

A concept lattice is used to provide insight in the configurations and their relations.

[Siff and Reps, 1997] and [Siff and Reps, 1998] describe the application of FCA to modularise procedural programs into classes. The formal context $C=(O,A,R)$ is used in the following way:

- O: set of functions in the source code.
- A: set of datatype definitions.
- R: uses-datatype and does-not-use relations. The following datatype usages are considered: return type, argument type and global variable type usage.

The algorithm generates the concept lattice and calculates concept partitions. A *concept partition* is a grouping of the concepts such that every atomic concept in the lattice appears

[10] This is based on [Snelting, 1996], [Tonella and Antoniol, 1999] and [Ball, 1999].

precisely once in the concept partition. Each partition represents a possible modularisation. It is up to the user to select the appropriate partition.

In the following paragraphs several other uses of FCA for program understanding are discussed. For more applications the interested reader is referred to [Tilley et al, 2003] and [Snelting, 2000].

4.2.3 Design pattern detection

[Tonella and Antoniol, 1999] describe the use of FCA to find recurring design constructs in object-oriented code. The key idea is that a design pattern amounts to a set of classes and a set of relations between them. Two different instances of a pattern have the same set of relations, but different sets of classes.
Let D be the set of classes in the design and T be the set of relationship-types between classes. For example $T=\{e,a\}$ defines the relationship types "extends" and "association". Then the set of inter-class relations P is typed $P \subseteq D \times D \times T$. To find pattern instances of k classes the formal context $C_k=(O_k,A_k,R_k)$ is used with:

- O_k: set of k-sized sequences of classes in the design. More precisely $O_k = \{(x_1,\ldots,x_k) \mid x_i \in D \wedge i \in [1..k]\}$ where k is called the *order* of the sequence.
- A_k: set of inter-class relations within the sequences in O_k. Each is a triple[11] $(x_i,x_j)_t$, where x_i and x_j are classes and t is a relationship-type. A_k is defined by $A_k = \{(i,j)_t \mid (x_i,x_j)_t \in P \wedge i,j \in [1..k]\}$.
- R_k: "possesses" relation between the elements in O_k and in A_k.

Figure 10 gives an example of a class diagram, for which Table 5 shows the corresponding set of labelled class relations (P) and a legend.

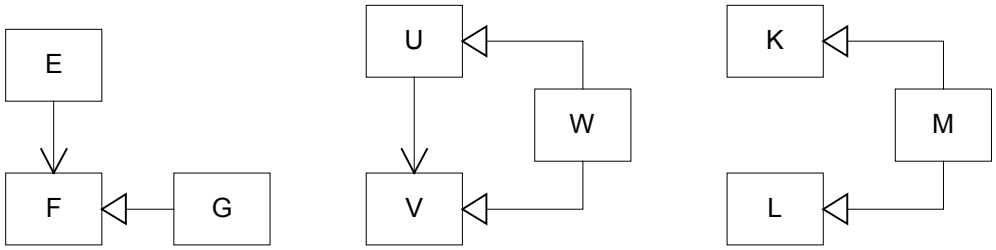
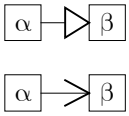


Figure 10: Example of a class diagram



$(\alpha,\beta)_e$

$(\alpha,\beta)_a$

$(E,F)_a$	$(U,V)_a$	$(M,K)_e$
$(G,F)_e$	$(W,U)_e$	$(M,L)_e$
	$(W,V)_e$	

Table 5: Set of labelled class relations *P*

A formal concept (X,Y) consists of a set of class-sequences X and a set of inter-class relations Y. Thus the intent Y specifies the pattern and the extent X specifies the set of pattern-instances found in the code.

Before the lattice can be constructed from the context this context must be generated from the class diagram. [Tonella and Antoniol, 1999] describe a simple inductive algorithm, which is shown in (11). Recall that D is the set of classes and P the set of class-relations.

The initial step generates an order two context. This is done by collecting all pairs of classes that are related by a tuple in P; the set O_2 of formal objects of the order two context consists

[11] We use the same notation as [Tonella and Antoniol, 1999].

of all pairs of classes related by a tuple in P. This means that for all formal objects in O_2 a relation of type t exists from the first to the second class. Therefore, the set A_2 of formal attributes of the order two context consists of the tuples $(1,2)_t$ for which a tuple in P exists that relates two arbitrary classes by a relation of type t.
In the inductive step, the order of the context is increased with one. The construction of O_k appends one component, x_k, to the tuples in O_{k-1}. This x_k is defined as any class for which a tuple in P exists that relates x_k to some other class x_j that is present in the tuple of O_{k-1}. Next, A_k is constructed by extending A_{k-1} with two sets of tuples. The first set consists of the tuples $(k,j)_t$, for which j equals the index of the class x_j that allowed the addition of x_k during the construction of O_k, and a relation of type t exists in P from x_k to x_j. The second set is similar, with k and j exchanged.

Initial step:

$$O_2 = \left\{ (x,y) \mid (x,y)_t \in P \right\}$$
$$A_2 = \left\{ (1,2)_t \mid \exists x,y \in D : (x,y)_t \in P \right\}$$

Inductive step $(k > 2)$:

$$O_k = \left\{ (x_1,\ldots,x_k) \mid (x_1,\ldots,x_{k-1}) \in O_{k-1} \wedge \exists j, 1 \le j \le k-1 \wedge \left((x_j,x_k)_t \in P \vee (x_k,x_j)_t \in P \right) \right\} \quad (11)$$
$$A_k = A_{k-1} \cup \left\{ (i,j)_t \mid \exists (x_1,\ldots,x_k) \in O_k \wedge \left((i = k \wedge 1 \le j \le k-1) \vee (j = k \wedge 1 \le i \le k-1) \right) \wedge (x_i,x_j)_t \in P \right\}$$

Note that in (11) the order n context contains the order $n-1$ context in the sense that all lower-order sequences are initial subsequences of the objects in the order n context, and that all attributes are retained. Note also that the algorithm assumes that design patterns consist of connected graphs. This assumption holds for all of the patterns in [Gamma et al, 1995], so provided that sufficient relationships between classes are extracted it does not impose a significant restriction.

Table 6 shows the order 3 context algorithm (11) generated for the example. The order 2 context contains for example the formal object (E,F). In the inductive step the tuple $(G,F)_e$ causes the extension to (E,F,G), and leads to the creation of a new formal attribute, $(3,2)_e$. Observe that the number of different formal objects is much less than the possible number of class combinations of length 3 (which is 9^3=729). This is due to the very low connectivity of the class-graph of the example (compared to a fully connected graph).

		Formal attributes A_3						
		$(1,2)_a$	$(1,2)_e$	$(3,2)_e$	$(3,2)_a$	$(3,1)_e$	$(2,3)_a$	$(1,3)_e$
Formal objects O_3	(E,F,G)	√		√				
	(G,F,E)		√		√			
	(U,V,W)	√		√		√		
	(W,U,V)		√				√	√
	(W,V,U)		√		√			√
	(M,K,L)		√					√
	(M,L,K)		√					√

Table 6: Order three context for pattern example

[Tonella and Antoniol, 1999] use algorithm (10) to construct the lattice. For the example this produces the concepts in Table 7 and the lattice in Figure 11.

c_t	({(E,F,G),(G,F,E),(U,V,W),(W,U,V),(W,V,U),(M,K,L),(M,L,K)}, ϕ)
c_0	({(E,F,G),(U,V,W)}, {$(1,2)_a$,$(3,2)_e$})
c_1	({(G,F,E),(W,V,U)}, {$(1,2)_e$,$(3,2)_a$})
c_2	({(U,V,W)}, {$(1,2)_a$,$(3,2)_e$,$(3,1)_e$})
c_3	({(W,U,V)}, {$(1,2)_e$,$(2,3)_a$,$(1,3)_e$})
c_4	({(W,V,U)}, {$(1,2)_e$,$(3,2)_a$,$(1,3)_e$})
c_5	({(M,K,L),(M,L,K),(W,U,V),(W,V,U)}, {$(1,2)_e$,$(1,3)_e$})
c_6	({(G,F,E),(W,U,V),(W,V,U),(M,K,L),(M,L,K)}, {$(1,2)_e$})
c_b	(ϕ, {$(1,2)_a$,$(1,2)_e$,$(3,2)_e$,$(3,2)_a$,$(3,1)_e$,$(2,3)_a$,$(1,3)_e$})

Table 7: Extents and intents of the pattern example

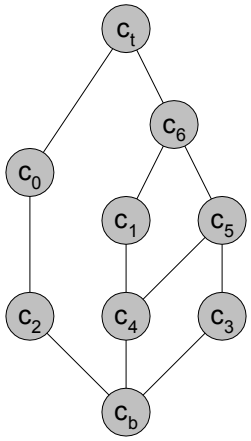


Figure 11: Concept lattice of the pattern example

The concepts in Table 7 directly represent patterns, but some redundancies are present. For example c_0 and c_1 represent the same pattern. [Tonella and Antoniol, 1999] informally define the notions of equivalent patterns and equivalent instances to remove redundancies from the lattice. (12) and (13) define these notions formally.

Definition 1 (Equivalent patterns): Let (X_1, Y_1) and (X_2, Y_2) be two concepts representing design patterns that are generated from the same order k context. (X_1, Y_1) and (X_2, Y_2) are equivalent patterns if an index permutation f on the index set $\{1..k\}$ exists such that:

$$X_2 = \left\{ \left(x_{f(1)},...,x_{f(k)} \right) \middle| \left(x_1,...,x_k \right) \in X_1 \right\} \wedge X_1 = \left\{ \left(x_{f^{-1}(1)},...,x_{f^{-1}(k)} \right) \middle| \left(x_1,...,x_k \right) \in X_2 \right\} \quad (12)$$

$(X_1, Y_1) \cong (X_2, Y_2)$ denotes that (X_1, Y_1) and (X_2, Y_2) are equivalent patterns.

According to Definition 1 two patterns (X_1, Y_1) and (X_2, Y_2) are equivalent when X_2 can be obtained by reordering the classes in (some of) the elements of X_1 and vice versa. Consequently, each formal attribute in Y_1 can be transformed into one in Y_2 and vice versa also.
In the example c_0 and c_1 are equivalent because the index permutation[12] $\{1 \rightarrow 3,\ 3 \rightarrow 1\}$ transforms $\{(E,F,G),(U,V,W)\}$ into $\{(G,F,E),(W,V,U)\}$ and vice versa.

[12] We use the informal notation $\{1 \rightarrow 3,\ 3 \rightarrow 1\}$ to refer to the index permutation $\begin{bmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{bmatrix}$.

Definition 2 (Equivalent instances): Let $(x_{1,1},\ldots,x_{1,k})$ and $(x_{2,1},\ldots,x_{2,k})$ be two formal objects in the extent X of an order k concept (X,Y) that represents a design pattern. These formal objects represent equivalent instances within that concept if an index permutation g on the index set $\{1..k\}$ exists such that:

$$\begin{aligned}\left(x_{2,1},\ldots,x_{2,k}\right)=\left(x_{1,g(1)},\ldots,x_{1,g(k)}\right)\wedge\left(x_{1,1},\ldots,x_{1,k}\right)=\left(x_{2,g^{-1}(1)},\ldots,x_{2,g^{-1}(k)}\right)\\ \wedge Y=\left\{\left(g\left(y_1\right),g\left(y_2\right)\right)_t\middle|\left(y_1,y_2\right)_t\in Y\wedge t\in T\right\}\end{aligned}\tag{13}$$

$(x_{1,1},\ldots,x_{1,k}) \cong (x_{2,1},\ldots,x_{2,k})$ denotes that $(x_{1,1},\ldots,x_{1,k})$ and $(x_{2,1},\ldots,x_{2,k})$ are equivalent instances.

According to Definition 2 two formal objects in the extent X of a concept (X,Y) are equivalent within that concept if an index permutation exists that transforms them into each other, and when applied to the formal attributes in Y produces attributes that are also part of Y.
In the example the set of formal objects of c_5 contains two pairs of equivalent instances because the index permutation $\{2\rightarrow3,\ 3\rightarrow2\}$ transforms (M,K,L) and (W,U,V) into (M,L,K) and (W,V,U) respectively, and $\{(1,2)_e,(1,3)_e\}$ into $\{(1,3)_e,(1,2)_e\}$.

If in the concepts in Table 7 all sets of equivalent patterns and equivalent instances are replaced with one representative element, and concepts with empty extents or intents (c_b and c_t) are removed, the concepts in Table 8 remain. Observe that concept c_6 is trivial; it represents the inheritance relation.

c_0	({(E,F,G),(U,V,W)}, {$(1,2)_a$,$(3,2)_e$})
c_2	({(U,V,W)}, {$(1,2)_a$,$(3,2)_e$,$(3,1)_e$})
c_5	({(M,K,L), (W,U,V)}, {$(1,2)_e$,$(1,3)_e$})
c_6	({(G,F,E),(W,U,V),(W,V,U),(M,K,L),(M,L,K)}, {$(1,2)_e$})

Table 8: Extents and intents of the pattern example

[Tonella and Antoniol, 2001] describe three case studies that apply the proposed method. Besides the static inter-class relations (inheritance and association), two other attributes are taken into account:

- Dynamic inter-class relations, for example the call and delegates relations.
- Class attributes such as member function definitions.

The method is applied to three public domain applications written in C++ (20-100 KLOC). [Tonella and Antoniol, 2001] report the detection of several recurring design constructs, including the Adapter pattern [Gamma et al, 1995] in several variants. The order of the context was chosen between two and four, typically three. Higher-order patterns did not prove to be a good starting point because *"they impose an increasing number of constraints on the involved classes and are therefore matched by few instances (typically just one)"* [Tonella and Antoniol, 2001]. For the order three context the number of formal objects was 1721 to 34147. The number of formal attributes was 10 in all cases. The construction of the concept lattice took between 1.8 and 85.8 seconds on a Sun SPARC 20 workstation[13].

4.2.4 Class structure analysis

[Dekel, 2002] describes the use of FCA to gain insight in the internal structure of a complex class. The method is based on the design heuristic that classes should have maximal *field-access class-cohesion*. The strongest version of field-access class-cohesion prescribes that all methods of a class should use all of its fields. The proposed method is based on the hypothesis that deviations from this rule represent a potential error in the internal structure.
In FCA terms, the context $C=(O,A,R)$ is used with:

- O: set of class member variables ("fields").
- A: set of class methods.
- R: method-uses-uses-field relation.

[13] The time to construct the context was not described in the paper.

In the lattice, the concepts represent groups of classes. A method-call graph is superimposed on the concept lattice to visualise the interactions between the methods.

The proposed method has been applied to Java classes. [Dekel and Gil] report its application to the Molecule class of the Java Chemistry Development Kit (75 public methods, 1500 LOC). [Dekel, 2002] describes the application to several versions the Graph class of the VGJ toolkit (Visualising Graphs with Java), which contained 43 to 69 methods and 5 to 9 fields. In all cases the methods provided insight in the structure of the class without overwhelming the user with the details typically found in source files.

4.2.5 Inheritance hierarchy analysis

[Arévalo and Mens, 2002] propose to use FCA to analyse how inheritance and interfaces relationships couple methods and classes in an inheritance hierarchy. The method takes the calling & delegation behaviour into account, but only within the inheritance hierarchy. Let c and d be classes and s a method signature. Then, in FCA terms the formal context $C=(O,A,R)$ is used with:

- O: set with method invocations from classes; $(c,s) \in O \Leftrightarrow$ some method in c calls s.
- A: set with classifications of the message sending behaviour [Arévalo and Mens, 2002]:
  - ConcreteSuperCaptureIn:d. (c,s) satisfies this predicate if s is called via a super send in some method of c and the receiver method is implemented in d, which is an ancestor of c.
  - ConcreteSelfCaptureLocally:c. (c,s) satisfies this predicate if s is called via a self-send in some method of c and the receiver method is a concrete method in c.
  - AbstractSelfCaptureLocally:c. (c,s) satisfies this predicate if s is called via a self-send in some method of c and the receiver method is defined as an abstract method in c.
  - ConcreteSelfCaptureInAncestor:d. (c,s) satisfies this predicate if s is called via a self-send in some method of c and the receiver method is defined as a concrete method in d that is an ancestor of c.
  - ConcreteSelfCaptureInDecendant:d. (c,s) satisfies this predicate if s is called via a self send in some method of c and the receiver method is defined as a concrete one in d that is an descendant of c.
- R: applicability of the message classifications in A to the invocations in O.

[Arévalo and Mens, 2002] describe the application of the proposed method to the Magnitude hierarchy (Smalltalk, 29 classes). The formal context was extracted from the code with the Soul logic programming language (part of FAMOOS). This produced 248 formal objects and 73 formal attributes. The application of FCA unveiled several common constructs.

The experiment also revealed a weakness of the approach; instead of only the receiver, the sender of a self-send should also be taken into account. [Arévalo, 2003] describes an improvement to the method that achieves this. If c is a class, m a method and s a method selector then $(c,m,s) \in O \Leftrightarrow$ method m in c calls s. The formal attributes are adapted accordingly. For details the interested reader is referred to [Arévalo, 2003].

The modified method has been applied in several case studies, which led to three important conclusions:

- The method is suitable to find unpredictable relationships.
- The method can be used to find behavioural patterns in inheritance hierarchies.
- The quality of the results depends on the chosen formal attributes. If the properties are too generic the method produces a few concepts that are not interesting enough. If the properties are too specific the method produces a lot of concepts with small extents.

4.2.6 X-Ray views

[Arévalo et al, 2003] describe the use of FCA to gain insight in the collaborations within a single class, similar to [Dekel, 2002]. The main difference between the two is that [Arévalo et al, 2003] incorporate dynamic behaviour into the concept lattice, whereas [Dekel, 2002] *superimposes* a call-graph on a concept lattice that is based on member-accesses.

The described approach uses FCA to gain insight in the elementary collaborations between class attributes and methods. Three types of relations are extracted from the source code, namely read-attribute, write-attribute and calls-method. Further, several indirect relationships are inferred from these:

- Attribute-access (read or write).
- Transitive versions of calls-method, reads-attribute, writes-attribute and accesses-attribute. The latter three consist of a sequence of calls-method relations, followed by a read-attribute, write-attribute and accesses-attribute relation.
- The complements of the preceding relationships.

The sets of extracted and inferred relationships are used to create two formal contexts $C_1=(O,A_1,R_1)$ and $C_2=(O,A_2,R_2)$ where:

- O: set of class methods.
- A_1: set of possible read or write accesses to class attributes. For example $A_1=\{reads\text{-}x, reads\text{-}y, writes\text{-}y\}$, in which x and y are class attributes.
- R_1: method-accesses-attribute relation between methods in O and elements of A_1.
- A_2: set of possible class-method invocations.
- R_2: method-calls-method relation between methods in O and elements of A_2.

After extending the above definitions to sets [Arévalo et al, 2003] define a number of high-level collaborations, which in turn are used to define three x-ray views of a class. The class and attribute relationships are extended to sets in two ways:

- $F\ R\ G$: R relates each entity in F to each one in G.
- $F\ \underline{R}\ G$: R exclusively relates each entity in F to each in G. This means that no pair of entities f,g exists such that $f \in F \wedge f\ R\ g \wedge g \notin G$ and inversely.

With these notions the following high-level collaborations are defined. For the sake of compactness they are described informally here. A more precise description can be found in [Arévalo et al, 2003].

- **Direct accessors**: set of methods with non-exclusive access to class attributes.
- **Exclusive direct accessors**: set of methods with direct access to class attributes by exclusive relationships.
- **Exclusive indirect accessors**: set of methods that call direct accessors.
- **Collaborating attributes**: sets of attributes that are used exclusively by a set of methods.
- **Statefull core methods**: set of methods that access all the state-defining attributes.
- **Collaborating methods**: set of methods that use the behaviour defined in the class.
- **Interface methods**: set of methods that are not used by the class itself.
- **Externally used state**: subset of the interface methods that directly access class attributes.
- **Stateless methods**: complement of the set of collaborating methods, i.e. set of methods that provide a service without calling other methods or accessing class attributes.

Using the above collaborations [Arévalo et al, 2003] define three x-ray views. Each view shows a different set of inner-class collaborations:

- **State usage** focuses on how methods access the state of classes. This view shows the exclusive direct accessors, exclusive indirect accessors, collaborating attributes and statefull core methods collaborations.
- **External/internal** categorises class methods according to their usage; internal or external. This view shows interface methods and externally used state collaborations.
- **Behavioural skeleton** focuses on how methods invoke each other within the class. This view shows the collaborating methods and stateless methods collaborations.

The proposed approach has been implemented in the ConAn tool, which is built on top of the FAMOOS framework. To validate the approach it has been applied to three Smalltalk classes from the VisualWorks distribution, namely OrderedCollection, UIBuilder and Scanner (3-18 attributes, 24-122 methods). [Arévalo et al, 2003] conclude that the approach allows *"iterative*

application of the defined views and opportunistic code reading". The fact that inheritance relationships are not taken into account is an important limitation of the approach.

4.2.7 Feature allocation analysis

[Eisenbarth et al, 2001] use a combination of FCA and static analysis techniques to build a mapping between functional, externally visible features of a program and relevant parts of the source code. FCA is used to locate the most feature-specific subprograms among a set of executed subprograms. The static analysis is used to retrieve a dependency graph of the program. Subprograms in this graph that are called by the subprograms found with FCA are added to the set of subprograms involved in the features.

More precisely, the method works are follows:

1. A set of relevant features $F=\{f_1, ..., f_n\}$ is identified.
2. A set of scenarios $A=\{S_1, ..., S_q\}$ is identified such that the features in F are covered.
3. For each scenario in A execution summaries are collected that list all subprograms executed during a run. This yields a set of required subprograms $O=\{s_1, ..., s_p\}$ for each scenario.
4. A relation table R is created such that $(S_1,s_1),\ (S_2,s_2),\ ...,\ (S_q,s_p) \in R$.
5. Concept analysis is applied to the context (O,A,R), producing a set of subprograms P that are associated with the features.
6. Dominance analysis and strongly connected component analysis is used to eliminate general-purpose subprograms that do not contain any feature-specific logic.
7. Static dependency analysis techniques such as program slicing are used to extract the code implementing the feature and all necessary variable and type declarations.

The proposed method is implemented using the Bauhaus toolkit and has been applied to two web browsers, Mosaic and Chimera (51 and 38 KLOC respectively). These two programs consisted of 701 and 928 subprograms respectively. The parts of the architecture relevant to two use cases were recovered, successfully unveiling a view of the architecture.

4.2.8 Framework usage analysis

[Viljamaa, 2002] describes the use of FCA to recover the reuse interfaces of object-oriented frameworks. The method searches for *specialisation patterns*, which are program structures that can be instantiated in several contexts. A specialisation pattern defines a set of roles that are played by structural elements of an instantiation. Knowledge about the most important specialisation patterns of a framework helps developers use the framework efficiently.

The types of roles that are chosen to analyse determine the formal context to which FCA is applied. If for example class roles are extracted the classes and their interfaces are selected as formal objects, and class features (e.g. inheritance relationships, declared methods and data fields) are used as formal attributes. To reduce the size of the context and prevent performance problems, the relevancy of program elements is checked before they are added to the context. A relevancy function r is associated with each program element v. Only those elements where $r(v) \geq 0$ are used as formal objects. FCA is then applied to the produced context, producing a concept lattice from which the specialisation patterns are extracted. This is accomplished by looking for contexts in which each program element plays precisely one role. This means each formal object must belong to one exactly extent. The context selection is implemented by calculating concept partitions from the lattice and selecting the appropriate one(s). A *concept partition* is a set of concepts whose extents form a partition of all formal objects in the formal context. So in the concept partition each formal object is part of one concept.

The proposed method is implemented in the Fred (FRamework EDitor) programming environment [Viljamaa, 2002] and is applied to reverse engineer the interface of the JUnit framework. The sources of the framework (50 classes) and a set of sample applications are used as input. [Viljamaa, 2003] reports that Fred found about half of the specialisation patterns used in the samples. The scalability of the used algorithms was reported to be a major obstacle.

4.2.9 Delfstof: detecting source code regularities

[Mens and Tourwé, 2004] propose the use of FCA to detect source code regularities in object-oriented code based on naming conventions. The formal context $C=(O,A,R)$ is used with:

- O: set of instances of source code entities like classes, methods and parameters.
- A: set of substrings of the names of the source code entities. Method- and class-names are split according to the capitals and other separators occurring in them. Small substrings and substrings with little conceptual meaning are discarded.
- R: containment relation between the entities in O and the substrings in A.

The substring comparison is case-insensitive, ignores colons, and reduces plurals to singulars.

[Mens and Tourwé, 2004] give an informal description of a filter that is applied to the produced concept lattice. (14) formally defines this filter. Let $c : O \rightarrow \{true,false\}$ be a function such that $c(x) \equiv true \Leftrightarrow x$ is a class (for every $x \in O$). Further, let $h(x)$ be the hierarchy entity x is part of. Then all concepts (X,Y) that satisfy

$$|X| \leq 2 \vee |Y| = 1 \vee \langle \forall x, y \in X : c(x) \wedge c(y) \wedge h(x) = h(y) \rangle \quad (14)$$

are discarded from the lattice. So the filter discards all concepts that are too small to be of interest or only contain classes in the same hierarchy.

After the filter has been applied the concepts in the lattice are classified into three categories:

- **Single class concepts** group concepts from which all elements belong to a single class.
- **Hierarchy concepts** group entities that belong to multiple classes in a single class hierarchy.
- **Crosscutting concepts** group entities from at least two class hierarchies.

Within the groups the lattice ordering is preserved whenever possible.

The proposed method is implemented in the Delfstof tool, which has been applied to five Smalltalk programs (52-271 classes). Several cases of copy-paste code reuse were detected, as well as several design pattern instances (detected through naming conventions).

5 Case study: Pattern detection

This chapter describes one of the two case studies discussed in this thesis. This case study investigates the detection of unknown structural design patterns in source code, using the method described in paragraph 4.2.3.

5.1 Case study goals

Design patterns capture design experience in the form of frequently used and proven design constructs for a certain context [Alexander, 1979]. Knowledge of applied design patterns helps maintainers understand the structure of a program and its rationale [Gamma et al, 1995], and is therefore useful for software maintenance[14].

[Kersemakers, 2005] used a pattern library to detect instances of structural design patterns in the source code of two subsystems of the Océ Controller. This way a view of the as-built architecture of these subsystems was reconstructed. A disadvantage of this approach is that it requires upfront knowledge on the implemented patterns. Furthermore, it suffers from variations in the implementation of the patterns.

The work described in paragraph 4.2.3 suggests that Formal Concept Analysis (FCA) can be used to find frequently used design constructs in source code without requiring upfront knowledge. Based on this, and the experiences of [Kersemakers, 2005], we formulate the following hypothesis:

> *H1: With Formal Concept Analysis frequently used structural design constructs in the source code of the Océ Controller can be detected without upfront knowledge on the expected structures.*

The confirmation of H1 does not imply that the found design constructs represent a *useful* architectural view of the Océ Controller. We therefore formulate an additional hypothesis:

> *H2: Knowledge of frequently used structural design constructs found with Formal Concept Analysis in the Océ Controller provides an architectural-view that is useful to gain insight in the structure of the system.*

The usefulness of knowledge on structural design constructs depends on the amount of information this knowledge gives. The number of classes in the pattern and the number of instances of the pattern are two important criteria for this. On average, the design patterns in [Gamma et al, 1995] contain about four to five classes. Because we are reconstructing an architectural view and not a subsystem-design we want to find slightly larger patterns. Hence we decided the patterns must contain at least six classes to be useful for architecture reconstruction.
The other criterion, the minimal number of instances of a useful pattern, is difficult to quantify. To our knowledge no work is published on this subject, so we determine it heuristically. Because no pattern-library is used, maintainers need to invest time to understand the patterns before reaping the benefit of this knowledge. The benefit, easier program understanding, must outweigh this investment. Obviously this is not the case if the patterns have one instance. Because we search repeated structures and not named patterns (like library-based approaches do) the investment is relatively high. Hence we decided that a pattern must have at least four instances to be useful to reconstruct an architectural view of the Océ Controller.

To confirm these two hypotheses a prototype has been built that implements the approach Tonella and Antoniol proposed, which is described in paragraph 4.2.3. Before applying the prototype to the complete Océ Controller it has been applied to two of its subsystems, namely Grizzly and the RIP Worker. Because this produced unsatisfactory results it was decided not to apply the prototype to the entire Océ Controller. For more information on the results of this case study the reader is referred to paragraph 5.4.

[14] See paragraph 2.4.3 for more information on this subject.

5.2 Pattern detection architecture

This paragraph describes the architecture of the pattern detection prototype. This architecture is based on the pipe and filter architectural style [Buschmann et al, 1999]. The processing modules have been implemented with two third party tools and XSLT transformations [XSLT, 2005].

XSLT is chosen because:

- **Functional programming**: XSLT allows functional programming. This is an advantage because one of the most important algorithms of the implemented approach is defined inductively (by (11)). This definition maps very well to a functional implementation.
- **Easy integration**: The two third-party tools, Columbus and Galicia, both support XML [XML, 2005] ex- and import.
- **Maturity**: XSLT is a mature and platform independent language.

Figure 12 shows a view of the prototype's architecture. The blocks represent processing-modules and the arrows directed communication channels between the modules. The latter are implemented with files.

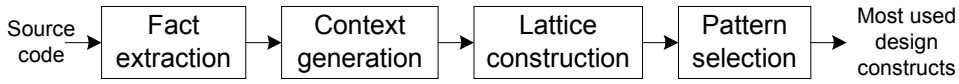


Figure 12: Architectural view of the prototype

The following paragraphs discuss each of these modules.

5.2.1 Fact extraction

The fact extraction module is based on the approach chosen by [Kersemakers, 2005]. In this step Columbus/CAN is used to extract structural information from the source code. Columbus uses the compiler that was originally used to compile the analysed software, in this case Microsoft Visual C++ [MSVC, 2005]. The extracted information is exported from Columbus with its UML exporter [Columbus, 2003], which writes the information to an XMI file. The schema Columbus uses can be found in [Columbus, 2003]. For more information about Columbus the interested reader is referred to paragraph 3.4.1.

Relationship types

Because the XMI file has a relatively complex schema the fact extraction module converts it to an XML file with a simpler schema. This file serves as input for the context generation module. It contains the classes and most important relationships between them.

Three types of relations are extracted [Booch et al, 1999]:

- **Inheritance**: The object-oriented mechanism via which more specific classes incorporate the structure and behaviour of more general classes.
- **Association**: A structural relationship between two classes.
- **Composition**: A special kind of association where the connected classes have the same lifetime.

Fact extraction output

Appendix 2 gives the schema of the XML file the fact extraction module produces. To illustrate the format we give a simple example here.
Figure 13 shows a class diagram with five classes, A up to F. The bracketed numbers are unique identifiers of the classes that are generated by Columbus. In the figure classes A and D inherit from B and E respectively. Classes B and D have an association with classes C and F respectively.

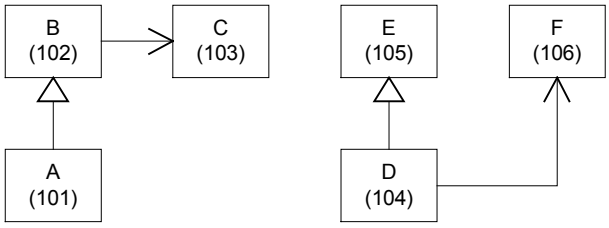


Figure 13: Example static structure

(15) shows the output of the fact extraction module for the class diagram in Figure 13. It consists of a *Model* containing *Classes* and *Relations* between classes. Each class has a *name* and a unique *id*, which is the number shown between the brackets in Figure 13. The *Relations* element contains *A* and *I* elements, which represent association and inheritance relations respectively. In case of the *A* element the *C1* and *C2* attributes contain the *id* of respectively the source and the destination class of the relation. In case of the *I* element the *C1* and *C2* attributes contain the *id* of respectively the child and the parent class. Composition relations are represented by a *C* element in the *Relations* element (not shown in (15)), using the same notation as the *A* element.

```
<Model>
    <Classes>
        <Class id="101" name="A" />
        <Class id="102" name="B" />
        <Class id="103" name="C" />
        <Class id="104" name="D" />
        <Class id="105" name="E" />
        <Class id="106" name="F" />
    </Classes>
    <Relations>
        <I C1="101" C2="102" />
        <A C1="102" C2="103" />
        <I C1="104" C2="105" />
        <A C1="104" C2="106" />
    </Relations>
</Model>
```
(15)

5.2.2 Context generation

This module uses the inductive context construction algorithm given in (11) to generate the formal context that will be used to find frequently used design constructs.

Recall that this algorithm consists of an initial and an inductive step. In the initial step an order two context is created. In the inductive step the order of the context is increased with one. This step is repeated until the desired order is reached. Also recall that the order of the context represents the number of classes in the patterns searched for.

Removing duplicates

Since XSLT does not support sets the prototype uses bags. This however allows the existence of duplicates. The prototype removes these with an extra template that is applied after the templates that implement each of the initial- and inductive steps. This produces the XSLT equivalent of a set.

Context generation output

After algorithm (11) has been completed, the “context generation” module converts the formal context to the XML import format Galicia uses for “binary contexts”. Appendix 3 gives the

schema of this XML file. To illustrate the format (16) gives a simple example for an order three context, which is based on the facts in (15).

In (16) the *BIN* element represents the binary context as a whole. Its *nbAtt* and *nbObj* attributes contain the number of formal-attributes and -objects in the context. The *type* attribute specifies the type of the context, in this case a binary context.
The *OBJS* element contains the formal objects. Each *OBJ* element has an identifier (*id*), and contains a string with the class IDs of the formal object separated by underscore characters.
The *ATTS* element contains the formal attributes. Each *ATT* element has an identifier (*id*) and contains a string with a relationship type and two *indices* in the formal objects, separated by underscore characters. These indices refer to positions in the formal objects. For example the *<ATT id="0">att_I_C1_C2</ATT>* element refers to the first and the second position.
The *RELS* element represents the relations between the formal objects in *OBJS* and the formal attributes in *ATTS*. It consists of *REL* elements, which specify that a certain formal object has a certain formal attribute through its *idObj* and *idAtt* attributes.

```
<BIN name="Example" nbAtt="6" nbObj="4" type="BinaryRelation">
    <OBJS>
        <OBJ id="0">obj_101_102_103</OBJ>
        <OBJ id="1">obj_102_103_101</OBJ>
        <OBJ id="2">obj_104_105_106</OBJ>
        <OBJ id="3">obj_104_106_105</OBJ>
    </OBJS>
    <ATTS>
        <ATT id="0">att_I_C1_C2</ATT>
        <ATT id="1">att_A_C1_C2</ATT>
        <ATT id="2">att_A_C2_C3</ATT>                                    (16)
        <ATT id="3">att_I_C3_C1</ATT>
        <ATT id="4">att_A_C1_C3</ATT>
        <ATT id="5">att_I_C1_C3</ATT>
    </ATTS>
    <RELS>
        <REL idObj="0" idAtt="0" />
        <REL idObj="0" idAtt="2" />
        <REL idObj="1" idAtt="1" />
        <REL idObj="1" idAtt="3" />
        <REL idObj="2" idAtt="0" />
        <REL idObj="2" idAtt="4" />
        <REL idObj="3" idAtt="1" />
        <REL idObj="3" idAtt="5" />
    </RELS>
</BIN>
```

For example in (16) the *<OBJ id="0">obj_101_102_103</OBJ>* element refers to the class sequence [A,B,C], using the IDs of the classes specified in (15). This sequence is constructed as follows. The initial step produces the string “obj_101_102” because the relation-element *<I C1=”101” C2=”102”>* exists. This relation also leads to the creation of the *<ATT id="0">att_I_C1_C2</ATT>* and *<REL idObj="0" idAtt="0" />* elements. The next inductive step extends the formal object to “obj_101_102_103” because the relation-element *<A C1=”102” C2=”103”>* exists and “102” is already part of the formal object. Now the complete *<OBJ id="0">obj_101_102_103</OBJ>* element has been obtained. This last extension also causes the creation of the *<ATT id="2">att_A_C2_C3</ATT>* and *<REL idObj="0" idAtt="2" />* elements.
The other elements in (16) are constructed similarly.

Size of the output

The initial step of the context generation algorithm produces an order two context. Each inductive step extends the order with one. So in general the $(k\text{-}1)$-th step of the algorithm $(k \geq 2)$ produces a context $C_k=(O_k,A_k,R_k)$ of order k, where O_k is the set of formal objects, A_k the set of formal attributes, and R_k the set of relations between the formal objects in O_k and the formal attributes in A_k.

The number of formal attributes, $|A_k|$, is bounded by the number of different triples that can be made. Each formal attribute in A_k is a triple (p,q,t) where p and q are integer numbers between 1 and k, and t is a relationship-type. The number of permutations of two values, each between 1 and k, is bounded by k^2 so at most k^2 different combinations are possible for the first two components of the formal attributes. Therefore, if T is the set of relationship-types, and the size of this set is $|T|$, $|A_k| \leq |T| \cdot k^2$.

The number of formal objects, $|O_k|$, in the order k context is limited by the number of permutations of different classes of length k. If D is the set of classes, and $|D|$ the size of this set, this means that $|O_k| \leq |D|^k$. So the number of formal objects is polynomial with the number of classes and exponential with the size of the patterns searched for. However, the fact that the connectivity of the classes in D is usually relatively low (and even can contain disconnected subgraphs), limits $|O_k|$ significantly.

Computational complexity

Let $P \subseteq D \times D \times T$ be the set of relations between classes, with D and T defined above. In the implementation the initial step is implemented with a template for the elements of P. Hence, if $|P|$ is the number of elements in P, the complexity of the initial step is $O(|P|)$.

The inductive step increases the order of the context with one. This is implemented with a template for the formal objects in the order $(k\text{-}1)$ context, so for the elements of O_{k-1}. This template extends each formal object $o \in O_{k-1}$ with a class that is not yet part of o and is related to one of the classes in o via a class-relation in P. Because every formal object in O_{k-1} consists of $k\text{-}1$ classes, the inductive step that produces O_k has a computational complexity of $O(|O_{k-1}| \cdot (k\text{-}1) \cdot |P|)$, which approximates $O(k \cdot |P| \cdot |O_{k-1}|)$.

Let $(x_1, \ldots, x_{k-1})$ be the sequence of classes represented by a formal object $o \in O_{k-1}$. Because in our implementation the previous inductive step appended classes to the *end* of this sequence[15], in the next inductive step only the last element x_{k-1} can lead to the addition of new classes to the sequence. Therefore, all but the first inductive steps do not have to iterate over all $k\text{-}1$ classes in the formal objects in O_{k-1}, but can only consider the most recently added class. This optimisation reduces the computational complexity of the inductive step to about $O(|P| \cdot |O_{k-1}|)$. Because of limited implementation time this optimisation has not been applied to the prototype however, but is left as future work.

Because $|O_{k-1}|$ is polynomial with the number of classes in D, and in the worst case $|P|$ is quadratic with $|D|$, this optimisation gives the inductive step a computational complexity that is polynomial with the number of classes in D. However, it is exponential with the size of the patterns searched for.

5.2.3 Lattice construction

The prototype constructs the lattice with a third party tool called Galicia. Galicia is an open platform for the construction, visualisation and exploration of concept lattices [Valtchev et al, 2003]. Its most important functions are the input of contexts, and lattice construction and visualisation [Galicia, 2005]. Galicia also implements interactive data inputs and various export formats.

[15] The fact that this is the *end* is not really relevant. The essential point is that the new class is always added at the same position.

Lattice construction algorithm

Galicia implements several algorithms to construct a lattice from a formal context. Based on their characteristics one of them is chosen for the prototype. [Kuznetsov and Obëdkov, 2001] compare a set of lattice construction algorithms, both theoretically and experimentally. They conclude that for large contexts the Bordat algorithm [Bordat, 1986] gives the best performance[16]. Because it is expected that the number of classes extracted from the source code, and hence the number of formal objects, will be relatively high, the Bordat algorithm is chosen to generate the lattice. Let L represent a concept lattice with $|L|$ formal concepts. Further, let $|O|$ and $|A|$ be the number of formal-objects and -attributes respectively of the formal context from which L is constructed. Then the Bordat algorithm has a worst-case computational complexity of $O(|O|\cdot|A|^2\cdot|L|)$.

Theoretically the size of the lattice, $|L|$, is exponential with the size of the context; if $|A|=|O|=n$ then $|L|\leq 2^n$. In practice however, the lattice-size may be $O(n)$ [Snelting, 1996], but this obviously depends on the properties of the formal context. When assuming that this is the case, and considering that in our case $|A|$ is much smaller than $|O|$, the computational complexity of the Bordat algorithm approximates $O(|O|^2)$. Recall that the number of formal objects is polynomial with the number of classes and exponential with the size of the patterns searched for. This means that the computational complexity of the lattice construction is polynomial with the number of classes in the source files and exponential with the size of the patterns.

Lattice construction output

Figure 14 shows the lattice Galicia produces for the formal context in (16). Each node in the graph represents a formal concept with its extent (E) and intent (I). The subconcept relations between the concepts[17] determine the structure of the graph. The numbers inside the nodes are the unique identifiers of the formal concepts Galicia assigned to them.

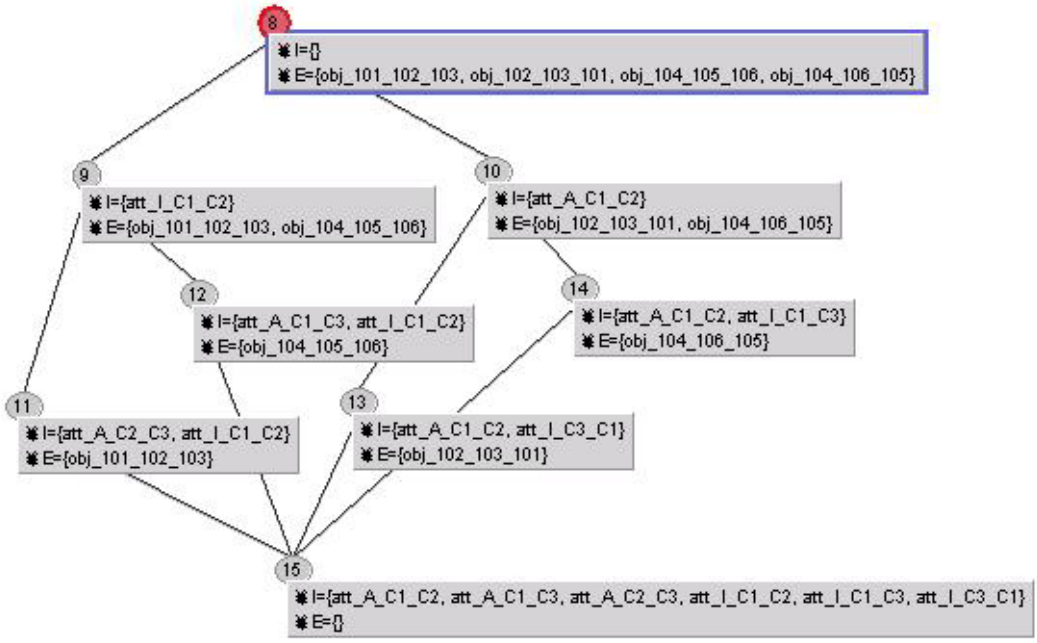


Figure 14: Galois lattice for formal context in (16)

[16] The following algorithms were examined: Bordat, Ganter, Close by One, Lindig, Chein, Nourine, Norris, Godin, Dowling, and Titanic. For details the reader is referred to [Kuznetsov and Obëdkov, 2001].

[17] The extent, intent and the subconcept relation are discussed in paragraph 4.2.1.

For example, concept 9 represents the class sequences [101,102,103] and [104,105,106], and has an intent that consists of the formal attribute $att\_I\_C1\_C2$. This concept represents a pattern with two instances in which the first class in the sequence inherits from the second.

The next module of the prototype filters the found formal concepts. Like the other data transformations in the prototype, this step is implemented with XSLT templates. The lattice is exported from Galicia in the XML format described in Appendix 4. (17) shows a fragment of this file that is produced for the formal context in (16). The $OBJS$ and $ATTS$ elements are the same as in (16) and are shown empty here. The $NODS$ element represents the set of formal concepts, each represented by a single NOD element. (17) shows only one such element; the actual XML file contains eight. The extent and intent of each NOD element are represented by the EXT and INT elements respectively, and contain references to the elements in $OBJS$ and $ATTS$ respectively. The SUP_NOD element describes the subconcept ordering of the lattice but is not relevant for the prototype.

```
<LAT type="LinkedConceptLattice">
    <OBJS/>
    <ATTS/>
    <NODS>
        <NOD id="9">
            <EXT>
                <OBJ id="0"/>
                <OBJ id="2"/>
            </EXT>
            <INT>
                <ATT id="0"/>
            </INT>
            <SUP_NOD>
                <PARENT id="8"/>
            </SUP_NOD>
        </NOD>
    </NODS>
</LAT>
```
(17)

5.2.4 Pattern selection

The final module of the prototype filters the patterns in the lattice. Two filters are applied. First, sets of equivalent formal concepts, in the sense defined by (13), are replaced by one of their elements. Second, the concepts are filtered according to the size of their extent and intent (the number of formal objects and attributes respectively). In the remainder of this paragraph these two filters are described more precisely

The prototype does not filter for equivalent patterns in the sense defined by (12). It was planned to add this later if the output of the prototype proved to be useful. However, as is described in paragraph 5.4, this was not the case.

Equivalent formal object filtering

Let X be the set of formal objects of some formal concept the lattice construction module produced, and let instance equivalence $\cong$ be defined by (13). Then, for every formal concept, the result of the first filter is the subset $X' \subseteq X$ that is the maximal subset of X that does not contain equivalent instances. If $|X'|$ and $|Z|$ refer to the number of elements in X' and another set Z respectively this is defined as:

$$\begin{aligned} &X' \subseteq X \wedge f(X') \wedge \neg\exists Z \subseteq X : f(Z) \wedge |Z| > |X'| \\ &\text{with } f(X') \equiv \neg\exists x_1, x_2 \in X' : x_1 \neq x_2 \wedge x_1 \cong x_2 \end{aligned} \tag{18}$$

This filter is implemented with two templates for the formal objects (the elements of X). The first template marks, for every formal concept, those formal objects for which an *unmarked*

equivalent instance exists. Of every set of equivalent instances this leaves one element unmarked. The second template removes all marked formal objects. It is easy to see that this produces the maximal subset of X that does not contain equivalent instances.

Let $avg(|X|)$ and $avg(|Y|)$ represent the average number of formal objects and formal attributes respectively of the formal concepts. If $|L|$ represents the number of formal concepts in the lattice, the first filter then has a time complexity of $O(|L| \cdot avg(|X|) \cdot avg(|Y|))$.

Size-based filtering

The second filter removes all formal concepts with a small number of formal-objects or -attributes. Let p_x and p_y be two user-specified parameters that specify the minimum number of required formal-objects and -attributes respectively. Then the output of this filter only contains concepts with at least p_x formal objects and p_y formal attributes.

This is implemented with a trivial template for the elements in the lattice. If $avg(|X'|)$ represents the average size of the formal objects after equivalent instances have been removed, and $avg(|Y|)$ and $|L|$ are defined in the previous section, this has a computational complexity of $O(|L| \cdot (avg(|X'|) + avg(|Y|)))$.

Total complexity of the pattern selection

The two filters are applied subsequently. Because $avg(|X'|)$ is smaller than $avg(|X|)$, the pattern selection module has a computational complexity of approximately $O(|L| \cdot avg(|X|) \cdot avg(|Y|))$.

We now express these three terms in terms of the number of formal objects. Recall that $|L|$ represents the number of formal concepts in the lattice and that we assume it to be proportional to the number of formal objects (and the number of formal attributes, but that is much less). If every formal attribute is associated with every formal object, $avg(|Y|)$ equals the number of formal objects. Because we assume the number of formal attributes to be very small compared to the number of formal objects, $avg(|X|)$ is not relevant for the computational complexity. Therefore, the computational complexity of the filtering module is approximately quadratic with the number of formal objects. Recall that the number of formal objects is polynomial with the number of classes and exponential with the size of the patterns searched for. This means that the complexity of the pattern-selection is polynomial with the number of classes in the input and exponential with the size of the patterns searched for.

5.3 Implementation validation

Before the prototype can be used to detect frequently used design constructs in source code, it must be ensured that the implementation is correct. This paragraph discusses how this has been handled.

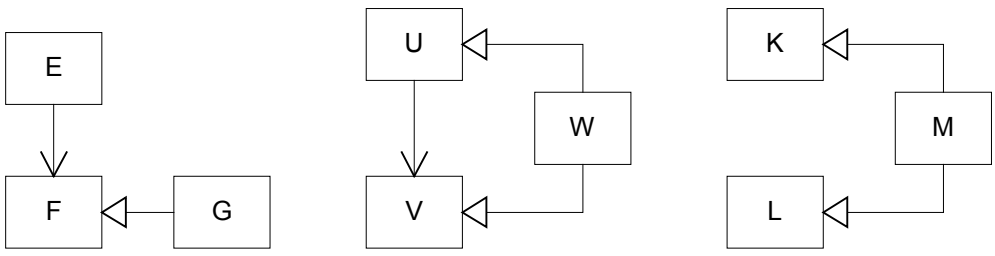


Figure 15: Validation code structure

To validate the quality of the prototype implementation, it is applied to a reference program written in C++. The structure of this program is the same as the example given in paragraph 4.2.3, whose structure is repeated in Figure 15. The squares represent classes and the relations are shown in UML notation [Booch et al, 1999]. For example class E has an association relation to class F and class W inherits from classes U and V.

Table 9 shows the output of the lattice construction module for the structure in Figure 15. Every row in the table represents a formal concept. Each formal concept is a tuple (X,Y), where X represents the set of formal objects and Y the set of formal attributes. The formal attributes are shown using the same notation as in paragraph 4.2.3; each is a triple $(p,q)_t$, where p and q represent indices in the formal objects, and t the type of the relation between the classes.

1	({(E,F,G),(U,V,W)}, {$(1,2)_a$,$(3,2)_e$})
2	({(G,F,E),(M,L,K),(W,U,V)}, {$(1,2)_e$})
3	({(U,V,W)}, {$(1,2)_a$,$(3,1)_e$,$(3,2)_e$})
4	({(G,F,E),(W,V,U)}, {$(3,2)_a$,$(1,2)_e$})
5	({(M,L,K),(W,U,V)}, {$(1,2)_e$,$(1,3)_e$})
6	({(W,V,U)}, {$(3,2)_a$,$(1,2)_e$,$(1,3)_e$})
7	({(W,U,V)}, {$(2,3)_a$,$(1,2)_e$,$(1,3)_e$})

Table 9: Prototype output

Observe that in Table 9 concept 4 can be transformed into concept 1 and vice versa with the index permutation {1→3, 3→1}. Therefore concept 4 and 1 are equivalent (according to definition (12)). Concepts 6 and 3 are also equivalent, as are 7 and 3.

Table 10 shows the output of the pattern-selection module resulting from the automatic filtering and the manual removal of redundant equivalent patterns from the concepts in Table 9. The two user-specified filtering-parameters are both set to one ($p_x=p_y=1$). Observe that the shown concepts are equivalent to the patterns in Table 8 (the pattern example). In fact, except for pattern 5, which is equivalent to c_5 in Table 8, the patterns are exactly the same. This confirms the correctness of the prototype's implementation.

1	({(E,F,G),(U,V,W)}, {$(1,2)_a$,$(3,2)_e$})
2	({(G,F,E),(M,L,K),(M,K,L),(W,U,V),(W,V,U)}, {$(1,2)_e$})
3	({(U,V,W)}, {$(1,2)_a$,$(3,1)_e$,$(3,2)_e$})
5	({(M,L,K),(W,U,V)}, {$(1,2)_e$,$(1,3)_e$})

Table 10: Prototype output after manual filtering

5.4 Results of pattern detection case study

The prototype has been applied to the Grizzly and RIP Worker subsystems of the Océ Controller. The characteristics of these subsystems have been given in paragraph 1.2. The following paragraphs give some examples of the found patterns. In all cases classes will be visualised as squares and the relations between them with UML notation [Booch et al, 1999].

5.4.1 Results for Grizzly

The application of the prototype to the Grizzly source code (234 classes) produced a formal context and a lattice with the characteristics shown in Table 11.

Number of formal objects	40.801
Number of formal attributes	37
Number of attribute-object relations	128.065
Number of formal concepts	989

Table 11: Characteristics of the order four context for Grizzly and the corresponding lattice

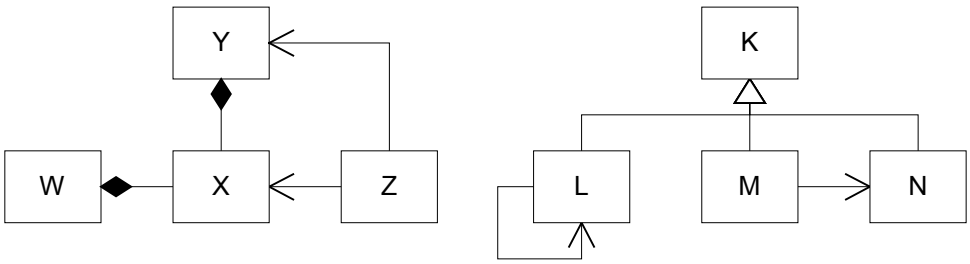
Recall from the "Size of the output" section in paragraph 5.2.2 that the number of formal attributes of an order k context, $|A_k|$, is bounded by the number of relationship-types, $|T|$, multiplied with k^2, so $|A_k| \leq |T| \cdot k^2$. In this case, $|T|=3$ and $k=4$ so the number of formal

attributes is bounded by $3 \times 4^2 = 48$. Observe in Table 11 that the number of formal attributes (37) is indeed less than 48.
Recall from the same section that the upper bound of the number of formal objects of an order k context, $|O_k|$, is polynomial with the number of classes $|D|$. More specific $|O_k| \leq |D|^k$. Since the characteristics in Table 11 are of an order four context, $|O_k| = 234^4 \approx 3.0 \cdot 10^9$, which is clearly more than 40.801. In fact, the number of formal objects is in the same order as $234^2 = 54.756$. This large difference is due to the low connectivity of the classes.

The figures in Table 11 confirm the assumptions made in paragraph 5.2.3. The number of formal attributes is indeed much lower than the number of formal objects. Furthermore, the number of formal concepts is not exponential with the size of the context. In fact, it is about one order smaller than the number of formal objects. This confirms our assumption in paragraph 5.2.3 that the size of the lattice is approximately linear with the number of formal objects.

With the user-specified filtering-parameters both set to four ($p_x = p_y = 4$), the prototype extracted 121 order four concepts from this context (with $p_x = p_y = 5$ only twelve remained). However, despite the filtering, many of the found patterns were very similar. The result even included several variants of the same pattern, for example with the associations organised slightly different.

The 121 concepts obtained with both filtering parameters set to four have been analysed manually according to their number of formal-objects and -attributes. Figure 17 shows two of the found patterns that were among the most interesting ones. For each pattern the ID of the corresponding formal concept is shown, as well as the number of formal-objects and -attributes. Galicia generated the concept-IDs, which uniquely identify the concept within the lattice.



Concept ID=678
Nr. of formal objects=20
Nr. of formal attributes=4

Concept ID=941
Nr. of formal objects=21
Nr. of formal attributes=5

Figure 17: Two patterns found in Grizzly

Concept 678 represents a pattern with classes W, X, Y and Z, where Z has an association with X and Y. Furthermore, both W and Y have a composition relationship with X. Analysis of the 20 instances of this pattern learns that for W fourteen different classes are present, for X and Y both two, and for Z three. This indicates that the instances of this pattern occur in a small number of source-code contexts.

Table 12 shows four example instances of this pattern. Examination of the Grizzly design documentation [Delnooz and Vrijnsen, 2003] learns that the first instance in Table 12, with W=BitmapSyncContext, covers a part of an Interceptor pattern [Buschmann et al, 1999]. This pattern plays an important role in the architecture of Grizzly. The BitmapDocEventDispatcher class plays the role of event Dispatcher, and the BitmapSyncContext the role of ConcreteFramework. The abstract and concrete Interceptor classes are not present in the detected pattern[18]. The EventDispatcherTest class is part of the Grizzly test code, and plays the role of the Application class in the Interceptor pattern. The Document class is not part of

[18] The designers of Grizzly omitted the abstract Interceptor class from the design.

the Interceptor pattern. In the Grizzly design this class is the source of the events handled with the interceptor pattern.
Observe that the pattern in Figure 17 does not contain the “create” relation between the BitmapDocEventDispatcher (Y) and the BitmapSyncContext (W) classes [Buschmann et al, 1999] specified. This does not mean that this relationship is not present; it is omitted from this pattern because the other pattern instances do not have this relationship.

W	X	Y	Z
BitmapSyncContext	Document	BitmapDoc EventDispatcher	BitmapDocEvent DispatcherTest
SheetDocEventDispatcher			
FlipSynchronizer	BasicJob	BitmapDoc Synchronizer	InversionWorkerJobInterceptor
StripeSynchronizer			BitmapDocSynchronizerTest

Table 12: Example instances of pattern 678

The other concept shown in Figure 17 (with ID 941) represents a relatively simple pattern with four classes labelled K, L, M and N. In this pattern class L, M and N inherit from K, L has a self-association, and M an association to N. As shown in Figure 17, 21 instances of this pattern are detected. Analysis of these instances of learns that in all cases K refers to the same class, L to three, and M and N both to six different classes. This indicates that all instances of this pattern are used in the same source-code context.

Table 13 shows four of the detected instances of pattern 941. SplitObjectStorage is an abstract class from which all workflow-related classes that store data inherit. The “SplitList” classes are container classes, for example for SplitTransition classes. The SplitTransition classes each represent a single state transition and are each associated with two SplitState objects. These represent the states before and after the transition.

K	L	M	N
SplitObjectStorage	SplitListOfAllTransitions	SplitTransition	SplitState
		SplitNode	SplitDoc
	SplitListOfAllStates	SplitState	SplitAttribute
	SplitListOfAllDocuments	SplitDocPart	SplitImageSequence

Table 13: Example instances of pattern 941

Surprisingly, the Grizzly design documentation [Delnooz and Vrijnsen, 2003] does not mention any of the classes listed in Table 13. Analysis of the code learns that these classes are concerned with workflow management in the Océ Controller, and represent points where Grizzly interfaces with the rest of the system. Strictly speaking these classes are not part of Grizzly but of the workflow-management subsystem of the Océ Controller. However, they are redefined in the Grizzly source-tree, and hence extracted by Columbus.

Observe that the two described patterns have a relatively low complexity. Recall that the two patterns described here are among the most interesting ones that are detected. So on average the complexity of the detected patterns is slightly lower that of the patterns described here.

5.4.2 Results for RIP Worker

Applying the prototype to the RIP Worker source code (108 classes) produced a formal context and a lattice with the characteristics shown in Table 14.

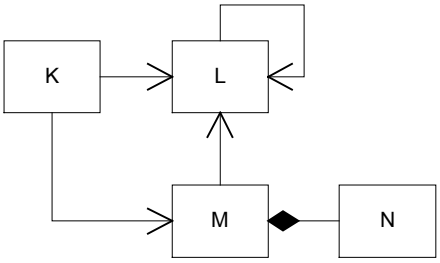
Number of formal objects	52.037
Number of formal attributes	41
Number of attribute-object relations	170.104
Number of formal concepts	3.097

Table 14: Characteristics of the order four context for the RIP Worker and the corresponding lattice

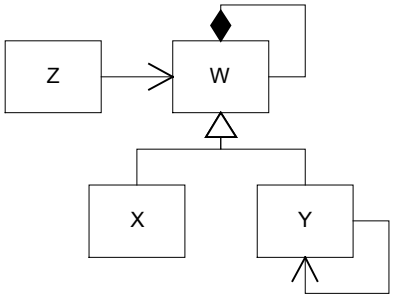
Observe that, if $|T|$ is the number of relationship-types, $|T| \cdot k^2$ is an upper bound of the number of formal attributes of the order k context. This confirms our assumption in the "Size of the output" section in paragraph 5.2.2. The number of formal objects of the order k context, $|O_k|$, does not exceed the upper bound predicted in the "Size of the output" section in paragraph 5.2.2. Table 14 represents an order four context, and $|O_k| = 52.037 \leq |D|^4 = 108^4 \approx 1,4 \cdot 10^8$, so the number of formal objects is relatively low. As with Grizzly, this is due to the low connectivity of the classes.

Observe also that the figures in Table 14 confirm our assumptions in paragraph 5.2.3: like with Grizzly, the size of the lattice is approximately linear with the size of the context (one order smaller), and the number of formal objects is much higher than the number of formal attributes.

With the user-specified filtering-parameters both set to five ($p_x = p_y = 5$), the prototype produced 158 order four concepts (with $p_x = p_y = 4$ 799). Like the patterns found in Grizzly, the set of patterns found in the RIP Worker also contains a lot of similar patterns. Figure 18 shows two of the found patterns, together with their number of formal-objects and -attributes.



Concept ID=2694
Nr. of formal objects=25
Nr. of formal attributes=5



Concept ID=2785
Nr. of formal objects=31
Nr. of formal attributes=5

Figure 18: Two patterns found in the RIP Worker

Concept 2694 represents a pattern with classes K, L, M and N, where class K has an association relationship with L and M, L a self-association, and M an association to L. Finally, class M has a composite relationship to N. Analysis of the output of the filtering-module learns that for class N 25 different classes are present, but for K, L and M all pattern instances have the same class. This indicates that all instances of this pattern are used in the same piece of the source code.

Table 15 shows four examples of pattern 2694. All are concerned with job-settings and the configuration of the system. The PJT_T_SystemParameters class stores information about the environment of the system, for example supported media-formats and -types. The PJT_T_JobSetting class represents the settings for a complete job, and is composed of the classes listed for N. The class listed for L, PJT_T_Product, is used to detect if the machine can handle a certain job-specification [DVRIP, 2002].

K	L	M	N
PJT_T_System Parameters	PJT_T_ Product	PJT_T_ JobSetting	PJT_T_MediaColor
			PJT_T_MediaWeight
			PJT_T_RunLength
			PJT_T_StapleDetails

Table 15: Example instances of pattern 2694

Concept 2785 represents a pattern with classes W, X, Y and Z, where X and Y inherit from W, Y has a self-association, and W a self-composition. Class Z is only loosely connected to the other classes, namely via an association to class W.
Analysis of the 31 instances of this pattern learns that in all cases W and Y refer to the same class. X refers to eight different classes and Z to four. This indicates that all instances of this pattern are used in the same source-code context.

Table 16 shows four example instances of pattern 2785. None of the listed classes are mentioned in the RIP Worker design documentation [DVRIP, 2002]. Examination of the source code learns that all instances are part of a GUI library the RIP Worker's test tools use.

W	X	Y	Z
CWnd	CDialog	CFrameWnd	CCmdUI
	CButton		CDialog
	CListBox		CWinThread
	CEdit		CDataExchange

Table 16: Example instances of pattern 2785

Similar to the result for Grizzly, the patterns described for the RIP Worker have a relatively low complexity. Since these patterns are the most interesting of the detected patterns, the other patterns can generally be regarded as uncomplicated.

5.4.3 Observations

Quality of the results

When examining the prototype's output for Grizzly and the RIP Worker it is clear that better filtering is required. Recall that filtering for equivalent patterns, as defined by (12), has not been implemented in the prototype. The output contains many equivalent patterns so in practice this filtering is desired too.

The occurrence of sets of patterns in the output with small differences represents a more significant problem. A possible filtering strategy might be to group highly similar patterns into subsets and (initially) show only one pattern of each subset of the user. This requires a measurement for the difference between patterns. This measurement could for example be based on the number of edges (class relations) that must be added and removed to convert one pattern into another. We leave this as future work.

After filtering the results manually, the remaining patterns are of a relatively low complexity, compared to for instance the patterns found in [Gamma et al, 1995]. More complex patterns typically have one instance and are removed by the pattern selection module. This means we are not able to achieve our goal of finding patterns that are useful to reconstruct architectural views (hypothesis H2).

In literature several publications report finding large numbers of design pattern instances in public domain code and few in industrial code, e.g. [Antoniol et al, 1998], [Kersemakers, 2005]. We speculate that it could be the case that industrial practitioners structurally design software in a less precise way than public domain developers. Obviously further experiments are needed to validate this statement, but it could explain why in our case study the number of instances of the found patterns remains fairly low.

Encountered problems

During the fact extraction process several problems were encountered. First of all, Columbus consistently crashed during the compilation of some source files. Recall that the source files are compiled with the same compiler as with which they were compiled during forward engineering ([MSVC, 2005]). Because they compiled without errors at that time, the error during fact extraction must either be caused by an incompatibility between Columbus and the Microsoft Visual C++ compiler, or by an error in Columbus itself.

This problem was encountered once while analysing the RIP Worker and ten times while analysing the full Océ Controller. In all cases, skipping the source file that triggered the error solved the problem. Because this only happened once for the RIP Worker, and not at all for Grizzly, this has little impact on the results described in paragraph 5.4.2.

The second encountered problem occurred during the linking step of the fact extraction. In this step the linker of Columbus combines the compiled source files, similar to the task of a linker during the generation of an executable. With the RIP Worker and Grizzly subsystems no problems were encountered, but with the complete Océ Controller Columbus crashed during this step. A few experiments revealed that this is probably caused by the size of the combined abstract syntax graphs, which is closely related to the size of the source files. Therefore it was not possible to extract facts from the full Océ Controller with Columbus.

Execution times

Both subsystems are analysed on the same test platform. Table 17 shows the characteristics of this platform.

Processor	Pentium 4, 2 GHz
Memory	2 GB
Operating system	Windows 2000 SP4
Columbus	3.5
Galicia	1.2
Java	1.4.2_06

Table 17: Test system characteristics

Table 18 shows the execution times for the RIP Worker and Grizzly subsystems for an order four context. All values, except for the lattice-construction time, are measured in wall-clock time. The lattice-construction time is measured in CPU time, but because the CPU load during this process was almost 100%, this is equivalent to wall-clock time. The time for lattice construction includes the time needed to import the formal context into Galicia and export the generated lattice to an XML file.
For Grizzly the total execution time was 7:44:59 and for the RIP Worker 11:17:17 (hh:mm:ss).

		Grizzly	RIP Worker
1	Fact extraction	0:01:09	0:42:40
2	Context generation	0:26:00	0:36:00
3	Lattice construction	4:41:50	6:57:37
4	Pattern selection	2:36:00	3:01:00

Table 18: Execution times (hh:mm:ss)

The patterns the prototype detected in the Grizzly and RIP Worker source code are relatively simple. Possibilities to produce more interesting patterns are:

1. Extending the size of the input to, for example, multiple subsystems of the Océ Controller.
2. Increasing the order of the context. This increases the number of classes in the patterns, and hence their complexity.
3. Introducing partial matches.

The third possibility, partial matches, requires fundamental changes to the method. If FCA would still be used, these changes would increase the size of the lattice significantly, and hence the execution time of the lattice construction step.

The first two options have the disadvantage that they increase the size of the data that is processed. This affects the running time of all modules. Recall that the computational complexity of the algorithms each of the modules uses is polynomial with the number of classes and exponential with the order of the context. Based on this, and the executing times

in Table 18, we concluded that, from a performance point of view it is not practical to use the prototype to reconstruct architectural views of the complete Océ Controller[19].

5.5 Conclusions of the pattern detection case study

This case study aimed to investigate the following hypotheses:

H1: With Formal Concept Analysis frequently used structural design constructs in the source code of the Océ Controller can be detected without upfront knowledge on the expected structures.

H2: Knowledge of frequently used structural design constructs found with Formal Concept Analysis in the Océ Controller provides an architectural-view that is useful to gain insight in the structure of the system.

Paragraph 5.1 describes two criteria for a structural design construct to be useful to reconstruct an architectural view; it must contain at least six classes and have at least four instances.

To confirm the two hypotheses, a prototype has been built that implements the approach [Tonella and Antoniol, 1999] proposed. This prototype has been applied to two subsystems of the Océ Controller, leading to the following conclusions:

- FCA can indeed be used to find frequently used design constructs in source code without upfront knowledge on the expected constructs.
- The performance our XSLT implementation of the approach is such that it is not feasible to analyse very large software structures. Although this is partly due to inefficiencies in our XSLT implementation, the computational complexity of the used algorithms is the main reason for this. Since applying the algorithms to two small subsystems of the Océ Controller already requires a lot of time we conclude that it is not practical to apply the approach to the complete Océ Controller. With a more efficient implementation it seems possible to detect design patterns in its subsystems though. These subsystems are about five to ten percent of the size of the total system.
- The found design constructs are of a limited complexity. For performance reasons no contexts of orders large than four could be analysed, so the detected patterns consisted of four classes or less. Although large numbers of pattern instances were detected, these were typically confined to a few areas of the source code.
- Due to an error in Columbus/CAN, this fact extractor cannot be used to extract facts from the complete Océ Controller.

This case study shows that finding patterns without a pattern library takes a lot of computing time, even for relatively simple patterns in relatively small pieces of software. Since it was possible to find frequently used design constructs, the results confirm hypothesis H1. Because it was not possible to detect patterns with six classes or more, we failed to confirm H2.

This leads to the conclusion that the prototyped approach is not (yet) useful to reconstruct architectural views of the complete Océ Controller. Using the distinction between architecture and design described in paragraph 2.4.3, we conclude that it can be used to reconstruct subsystem designs.

[19] The Océ Controller contains about ten to twenty times more classes than the two subsystems used in the experiment.

6 Clustering-based architecture reconstruction

The second of the two case studies described in this thesis uses clustering techniques to reconstruct an architectural view from source code. This chapter describes similar approaches reported in literature.

6.1 Clustering introduction

Before literature on clustering-based architecture reconstruction is discussed, this paragraph gives a non-exhaustive overview of clustering techniques. For more information on clustering the interested reader is referred to [Jain et al, 1999], [Berkhin, 2002] and [Pal and Mitra, 2004].

Clustering is a data analysis technique for dividing data elements into groups of similar elements that are called clusters [Berkhin, 2002]. This division is based on the similarity of data elements, which are usually represented as points in a multidimensional space or vectors of measurements [Jain et al, 1999]. Intuitively, in a valid clustering the data elements within a cluster are more similar to each other than to those in other clusters. Figure 19 shows an example of a clustering of points in a two-dimensional space.

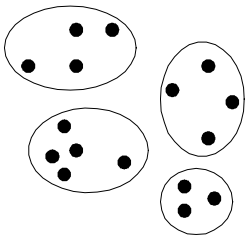


Figure 19: Example clustering

Various terms are used to refer to the data elements. Publications that describe the clustering process sec call them *objects* [Berkhin, 2002], [Pal and Mitra, 2004] or *patterns* [Jain et al, 1999]. [Lakhotia, 1996] presents a unified framework for software subsystem classification techniques where the data elements are called *nodes*. Approaches that use clustering for reverse engineering often use the terminology [Wiggerts, 1997] introduced, in which the clustered data elements are called *entities*. To avoid confusion with the object-oriented notions of objects and patterns we decided to use the latter term.

Clustering is an unsupervised classification technique. In general two types of classification techniques can be distinguished [Jain et al, 1999]:

- **Supervised** classification techniques start with a collection of pre-classified entities. The problem is to classify a newly encountered entity based on this collection. The pre-classified entities are often used to train the algorithm to recognise distinguishing characteristics of the entities, after which new entities can be classified.
- **Unsupervised** classification techniques do not start with a collection of pre-classified entities. Instead, they work solely on the collection of unclassified entities.

Clustering has many applications, including the classification of plants and animals, speech and character recognition, image segmentation, information retrieval and data mining. [Jain et al, 1999] give examples of the last four. Chapter 6 of this thesis describes a case study where clustering is used for architecture reconstruction. After completing our description of clustering in general, this chapter describes clustering-based architecture reconstruction approaches reported in literature.

6.1.1 Definitions

The following definitions regarding clustering are used in this chapter. These definitions are based on [Jain et al, 1999] and [Wiggerts, 1997].

- An *entity* x is a single data item used by the clustering algorithm. Typically it consists of a feature-vector of D measurements. Observation, object, datum and pattern are some of the synonyms for entity used in literature.
- The individual components of x are called *features*. In literature the term *attribute* is also used.
- D is the *dimensionality* of the entity.
- An *entity set* $H=\{x_1,\ldots,x_N\}$ denotes a set of N entities. The i-th entity is denoted x_i $(1 \le i \le N)$, and the j-th feature of x_i as $x_{i,j}$ $(1 \le j \le D)$. Some clustering algorithms view the entity set as an $N \times D$ entity matrix.

6.1.2 Components of a clustering task

A typical clustering task involves the following issues [Jain et al, 1999]:

1. **Entity representation and feature selection** involves the selection of the entities and the features. This often comprises of selecting the features that lead to the most effective clustering.
2. A **similarity metric** is a metric or quasi metric on the feature space that is used to quantify the similarity of two entities. The proximity of entities is usually measured with a distance function defined on pairs of entities.
3. **Grouping the entities** can be performed with many different algorithms. Traditionally these are divided into hierarchical and partitional algorithms [Berkhin, 2002]. Hierarchical algorithms produce a series of *nested* partitions by splitting or combining clusters. Partitional algorithms iteratively relocate entities between clusters to optimise a clustering criterion.
4. **Data abstraction** is an optional step in which a representation of the grouping result is created that is meaningful to the user. Typically this is done with a compact description of each cluster in terms of cluster prototypes or representative entities such as the centroid[20].
5. **Assessment of output** is an optional step that consists of an, often subjective, validation of the grouping result.

The following paragraphs discuss these five issues in more detail.

6.1.3 Entity representation & feature selection

The first issue in any clustering process is the selection of features and entities. An important goal of this step is to reduce the dimensionality of the feature space, while retaining the salient characteristics of the entities [Mitra and Pal, 2004]. Although no theoretical guidelines exist that suggest the appropriate entities and features for a specific situation [Jain et al, 1997], statistical measures can be used to evaluate the quality of a proposed feature selection [Mitra and Pal, 2004].

In general two types of features can be distinguished [Jain et al, 1997]:

- Quantitative features, for example:
  - Continuous values (e.g. the age or length of people).
  - Discrete values (e.g. the number of children in a family).
  - Interval values (e.g. the beginning and end of an event).
- Qualitative features, for example:
  - Nominal or unordered values (e.g. gender or colour).
  - Ordinal values (e.g. temperature classifications like “hot” and “cold”).

[20] The centroid of a solid object is its centre of mass. The centroid of a cluster is the point in the feature space that is the “average” of the points in the cluster. This point can be seen as the centre of gravity of the cluster.

[Wiggerts, 1997] uses a different classification, distinguishing two feature-types we will call inter- and inner-entity features:

- Inter-entity features describe relationships between entities. In this case the search space is considered as a graph in which the nodes represent the entities and the edges the relationships between them.
- Inner-entities describe each entity's score on the features.

Although these two classifications appear different, the relationship between them is easy to see. The inter-entity features can be considered unordered qualitative features by using a matrix that contains for each pair of entities the number of edges between them. Obviously the inner-entity features can be classified according to the classification of [Jain et al, 1997].

Based on their features, the similarity measure calculates the similarity of two entities, as is described in the next paragraph. This can be based on the values of the features (e.g. with continuous feature-values), but also on the presence or absence of features (e.g. with unordered feature-values). These two types are called distance measures and association coefficients respectively.

6.1.4 Similarity measures

A similarity measure calculates the similarity or dissimilarity of two entities. The clustering algorithm uses this measure to determine which entities must be placed in the same cluster.

Consider the case where inter-entity features are used. Recall that in this case the search space is considered as a graph where the nodes represent the entities and the edges relations between them. The similarity of two entities may be based on the edges between them, for example by counting them. This is an example of a similarity measure called an association coefficient. In case of directed edges, similarity measures may or may not take the direction into account. If different edge types are present each may be associated with a different weight, in which case the weights of the edges have to be summed.

Similarity measures produce a value bounded by 0 and 1, where 0 indicates no similarity at all, and 1 no difference. Some similarity measures calculate the dissimilarity $dis(x_i,x_j)$, of two entities x_i and x_j. In this case the similarity $sim(x_i,x_j)=1-dis(x_i,x_j)$.

Distance measures and association coefficients are two frequently used types of similarity measures [Berkhin, 2002], [Wiggerts, 1997]:

- **Distance measures** usually calculate the dissimilarity of two entities based on numeric features. Most distance measures are based on the Minkowski metric. Using the definitions introduced earlier in this chapter for two entities x_i and x_j this metric can be described as:

$$dis_p\left(x_i,x_j\right)=\left(\sum_{k=1}^{D}\left|x_{i,k}-x_{j,k}\right|^p\right)^{1/p}=\left\|x_i-x_j\right\|_p$$

  where $1\le p\le\infty$. The most popular distance measure, the Euclidean distance, is a special case of this metric with $p=2$ [Jain et al, 1999]. The Manhattan distance is another special case of the Minkowski metric, but with $p=1$ [Berkhin, 2002]. A disadvantage of the Minkowski metrics is the tendency of the largest scaled features to dominate the others. Normalising the features, or introducing weighting schemes can solve this [Jain et al, 1999].
- **Association coefficients** calculate the similarity of two entities based on the presence or absence of qualitative features. The following matrix is commonly used in literature to define association coefficients [Wiggerts, 1997]:

entity x_i \ entity x_j	1	0	
1	a	b	$a+b$
0	c	d	$c+d$
	$a+c$	$b+d$	

In this matrix x_i and x_j are two entities with binary features that are either absent (0) or present (1). The value of a represents the number of features that are present in both x_i and x_j, b represents the number of features present in x_i but absent in x_j et cetera. Differences between the various association coefficients are cause by different handling of 0-0 matches and different weighting of matches and mismatches [Wiggerts, 1997]. Popular association coefficients are the *Rand* and *Jaccard* indices, sim_R and sim_J respectively, that are defined as [Berkhin, 2002]:

$$sim_R\left(x_i, x_j\right) = \frac{a+d}{a+b+c+d} \qquad sim_J\left(x_i, x_j\right) = \frac{a}{a+b+c}$$

Observe that these two indices differ in the way 0-0 matches are handled. [Wiggerts, 1997] calls the Rand association coefficient the *simple matching coefficient*. The *Sørensen-Dice* coefficient is similar to the Jaccard coefficient in the handling of 0-0 matches, but assigns double weight to 1-1 matches [Anquetil and Lethbridge, 1999]:

$$sim_{SD}\left(x_i, x_j\right) = \frac{2a}{2a+b+c}$$

6.1.5 Grouping the entities

This task creates the actual clusters using one of the similarity measures described in the previous paragraph. Clustering algorithms can be divided in two groups, hierarchical and partitional algorithms. Algorithms of the first type produce a hierarchy of nested clusters, whereas algorithms of the second type produce a single partitioning of the entities. This paragraph describes these types, and the refinements that are listed below. This selection is based on the taxonomy of [Jain et al, 1999] and the architectural clustering approaches described in this chapter.

1. Hierarchical
   a. Single link
   b. Complete link
   c. Average link
2. Partitional
   a. Square error
   b. Graph theoretic
   c. Evolutionary

Before these types of algorithms are discussed in more detail, several crosscutting issues are discussed that affect all algorithm types. This discussion is based on [Jain et al, 1999].

- **Agglomerative vs. divisive** concerns the starting point the algorithm chooses. Agglomerative algorithms start with each entity in a singleton cluster and merge clusters until some stopping criterion is satisfied. Divisive algorithms start with all entities in a single cluster and split clusters until some stopping criterion is satisfied. Agglomerative algorithms are also called “bottom-up” and divisive algorithms “top-down” [Lakhotia, 1996].
- **Monothetic vs. polythetic** relates to the use of the features. Polythetic algorithms use all the features simultaneously in the similarity calculations. Monothetic algorithms do not do this and consider for example the features sequentially, using a different feature in each cycle of the algorithm.
- **Hard vs. fuzzy** relates to the output of the clustering. Hard clustering algorithms assign each entity to a single cluster. Fuzzy algorithms assign each entity to a set of clusters, each with a certain degree of membership.
- **Deterministic vs. non-deterministic** is important for algorithms that do not consider the entire search space but a subset of it. When run repeatedly, deterministic algorithms produce the same clustering, whereas non-deterministic algorithms produce different ones.

Hierarchical clustering algorithms

Hierarchical algorithms produce a series of nested clusters. Each iteration of the algorithm combines two clusters (agglomerative) or splits a single cluster (divisive). In both cases a

dendrogram is produced. A dendrogram can be visualised in a tree where the nodes represent clusters and the edges merge or split decisions taken by the algorithm. If the inner nodes are numbered such that the numbers increase monotonically from each child node to the parent node, levels in the clustering process can be identified.
Figure 20 (left) shows an example of a dendrogram. Slicing the dendrogram at a certain level gives a partitioning. In Figure 20 the dotted line in the dendrogram leads to the partitioning on the right.

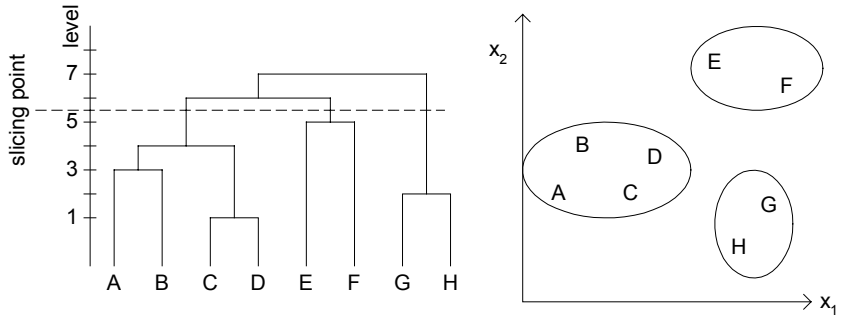


Figure 20: Hierarchical clustering example

Divisive hierarchical algorithms have the disadvantage that in the first iteration a large number of possible divisions must be evaluated. If the entity set consists of N entities, $2^{N-1}-1$ possibilities must be considered in the first iteration. Agglomerative algorithms do not have this disadvantage and are therefore the most widely used hierarchical algorithms [Wiggerts, 1997].

Agglomerative hierarchical algorithms all fit the following scheme (adapted from [Lakhotia, 1996] and [Wiggerts, 1997]):

$$
\begin{aligned}
&K = \text{set of clusters obtained by placing each entity in its own cluster}\\
&\text{Compute the similarities between the clusters in } K\\
&\text{while } |K|>1 \text{ do}\\
&\qquad K' = \text{subset of } K \text{ with most similar clusters such that } |K'|\geq 2\\
&\qquad K := K \setminus K' \cup merge\_clusters(K')\\
&\qquad \text{Update the similarities between the clusters in } K\\
&\text{od}
\end{aligned}
\tag{19}
$$

The algorithm starts with the creation of singleton clusters. The similarities between each pair of these clusters is computed by means of a similarity measure for entities, such as those discussed in the previous paragraph. In the first two steps of the while-loop, the algorithm finds the set of most similar clusters and merges them. In the third step of the loop the similarities are updated using one of the update rules described below. The algorithm iterates until a single cluster remains.

The last action of the while-loop, calculating the similarity between the newly formed cluster and the other clusters, can be implemented in several ways. All are based on the similarities of the merged clusters with the other clusters. These have been calculated earlier, either in the initialisation or a previous cycle of the loop. Let $k_a, k_b \in K$ be two distinct clusters that are merged. Then three popular update strategies to calculate the similarity between the merged cluster $k_a \cup k_b$ and a different cluster $k_c \in K$ are [Lakhotia, 1996]:

- Single link: $sim_{SL}(k_c,k_a \cup k_b)=sim_{SL}(k_c,k_a) \uparrow sim_{SL}(k_c,k_b)$
- Complete link: $sim_{CL}(k_c,k_a \cup k_b)=sim_{CL}(k_c,k_a) \downarrow sim_{CL}(k_c,k_b)$
- Average link: $sim_{AL}(k_c,k_a \cup k_b)=(sim_{AL}(k_c,k_a) + sim_{AL}(k_c,k_b))/2$

in which ↑ and ↓ denote the maximum and minimum of two values respectively. Two variants of the average link strategy are the weighted and unweighted average link strategies.
The complete link strategy produces compact or tightly bound clusters, whereas the single link strategy has a tendency to produce straggly or elongated clusters [Jain et al, 1999].

Figure 21 illustrates this (from [Wiggerts, 1997]). The dendrogram in Figure 20 is the result of applying an agglomerative hierarchical algorithm with a single link update strategy.

Figure 21: Clusterings obtained with single (left) and complete (right) link strategies.

According to [Jain et al, 1999] the time complexity of hierarchical agglomerative algorithms is typically $O(N^2 \log N)$, where N is the number of entities. Because agglomerative algorithms need to store a similarity matrix the space complexity is typically $O(N^2)$.

Partitional clustering algorithms

Partitional algorithms are the other main type of clustering algorithms. These algorithms produce a single partitioning of the entities and no hierarchy of clusters, as hierarchical algorithms do.

Due to the large number of possible combinations to cluster the entities it is usually not practical to try all of them. Partitional algorithms handle this by investigating only a part of the total search space, using various heuristical strategies. This typically causes these algorithms to converge at local optima. In practice the algorithm is often run multiple times with different starting states to handle this [Jain et al, 1999].

Several different types of partitional algorithms exist. In the remainder of this chapter three types will be discussed, namely square error, graph theoretical and evolutionary algorithms.

Square error clustering algorithms

Square error algorithms are the most frequently used partitional algorithms [Jain et al, 1999]. They start with an initial partition of the entities in a fixed number of clusters and iteratively relocate entities between clusters to optimise some clustering criterion. This criterion represents the quality of the clustering [Tzerpos and Holt, 1998].

Square error algorithms all use the following clustering criterion. Recall that H is the entity set, and let K be a clustering with $|K|$ clusters. Further, let c_j be the centroid of the j-th cluster and n_j the number of entities in the j-th cluster. Finally, let $x_i^{(j)}$ be the i-th entity belonging to the j-th cluster in K. Then the squared error criterion e^2 can be defined as [Jain et al, 1999]:

$$e^2(H,K) = \sum_{j=1}^{|K|} \sum_{i=1}^{n_j} \left\| x_i^{(j)} - c_j \right\|^2 \qquad (20)$$

In (20) $||x_i^{(j)}-c_j||$ represents a chosen distance measure between entity $x_i^{(j)}$ and the cluster centroid c_j. Squared error algorithms use the criterion in (20) to determine if some entity-relocation improves the quality of the clustering. The relocation is only applied if this is the case. When a certain convergence criterion is met the clustering process stops. Examples of convergence criteria are that the squared error value has not decreased for some number of iterations, or that no entity reassignment from one cluster to another has taken place for some number of iterations.

The k-means algorithm is the simplest and the most commonly used square error algorithm. In the initialisation-step a set of k cluster centroids is chosen, for example by randomly choosing k entities as cluster centroids, or by randomly choosing k points in the feature space. Next, the algorithm assigns the entities to the cluster that contains the closest centroid, after which the centroids are recomputed. The last two steps are repeated until a convergence criterion is met.

(21) describes this in pseudo-code (adapted from [Jain et al, 1999]):

$$
\begin{aligned}
&C = \text{set of } k \text{ randomly chosen cluster centroids} \\
&\text{repeat} \\
&\qquad K := \text{assign each entity to the cluster with the closest centroid in } C \\
&\qquad C := \text{centroids}(K) \\
&\text{while } \neg criterion(K)
\end{aligned}
\tag{21}
$$

k-means algorithms are sensitive to the initial choice of the cluster centroids, which can lead to the algorithm converging in a local minimum. Some variants of this algorithm attempt to choose initial centroids that are more likely to lead to a good clustering. Other variations of this algorithm permit the merging and splitting of clusters [Jain et al, 1999].

According to [Jain et al, 1999] the time complexity of the k-means algorithm is typically $O(N \cdot k \cdot l)$, where N is the number of entities, k the chosen number of clusters and l the number of iterations. If k and l are fixed in advance the algorithm has a linear time complexity. The space complexity is $O(k+N)$ because both the centroid-set and the entity-set need to be stored in memory.

Graph-theoretic clustering algorithms

Graph-theoretic algorithms are partitional algorithms that operate on graphs. The nodes of such graphs represent entities and the edges relations between these entities. In general graph algorithms try to split this graph into subgraphs that will form the clusters, instead of focussing on the entities themselves [Wiggerts, 1997].

The best-known graph-theoretic clustering algorithm uses a minimal spanning tree (MST) of the data [Jain et al, 1999]. A spanning tree is a graph connecting a set of N nodes such that a complete tree of N-1 edges is constructed. A spanning tree is minimal if the total length of the edges is the minimum necessary to connect all the nodes [Pal and Mitra, 2004]. Once the MST is constructed, the longest MST edges are deleted. The disconnected subgraphs obtained this way form the clusters [Jain et al, 1999].

Figure 22 shows an example of an MST. The numbers near the edges denote the length of the edges. The edge between the nodes labelled C and G (dotted in red) is the longest. If a single edge is removed to produce the clustering it will be this one, leading to the two clusters shown as ellipses in the figure.

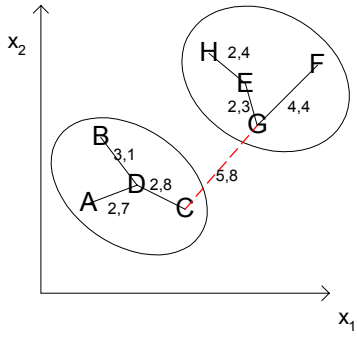


Figure 22: MST clustering example

Evolutionary clustering algorithms

Evolutionary algorithms are motivated by natural evolution. They use evolutionary operators and a population of solutions to obtain a globally optimal partition. Candidate solutions are encoded as 'chromosomes'. In evolutionary clustering algorithms these represent a partitioning. Examples of evolutionary operators are selection, recombination and mutation. Each of these transforms one or more input chromosomes into one or more output chromosomes.

Evolutionary clustering algorithms typically start with the generation of a random population of solutions, which represent a set of initial clusterings. The number of solutions in this population is called the *population size*. Each solution is associated with a fitness value that typically is inverse proportional to the squared error value of the corresponding clustering. Next, two steps are repeated until some termination condition is satisfied. First, the evolutionary operators are used to generate a new population of fitter solutions, typically with the same size as the previous population. Second, the fitness values are updated.
These steps can be summarized as follows (adapted from [Jain et al, 1999]):

$$\begin{array}{l} K := \text{random population of solutions.} \\ \text{Associate a fitness value with each solution in } K \\ \text{repeat} \\ \qquad K := next\_generation(K) \\ \qquad \text{Update fitness values} \\ \text{while } \neg termination(K) \end{array} \tag{22}$$

Genetic algorithms (GAs) are the best-known evolutionary clustering techniques [Jain et al, 1999]. These algorithms represent the solutions as binary strings, leading to a partitioning into two clusters. For example consider an entity set $H=\{x_1,...,x_6\}$ and the six bit binary string $[101110]$. This string corresponds to a clustering of the six entities in H in two clusters: $\{x_1,x_3,x_4,x_5\}$ and $\{x_2,x_6\}$.
The most popular recombination operator is the crossover operator [Jain et al, 1999]. This operator takes two solutions as input and swaps the substrings after the *crossover point*. Let $\otimes$ denote the crossover operator, p $(1<p<N)$ the crossover point, and $[s_1,...,s_N]$ and $[t_1,...,t_N]$ two binary strings that both classify N entities. Then the crossover operator applies the following transformation:

$$[s_1,...,s_N] \otimes_p [t_1,...,t_N] = \left([s_1,...,s_p,t_{p+1},...,t_N], [t_1,...,t_p,s_{p+1},...,s_N] \right)$$

For example $[011111] \otimes_2 [100010]=([10\ 1111],[01\ 0010])$.

The selection operator uses a fitness function that implements a similarity metric to select the best solutions. The mutation operator is used to reduce the possibility that the algorithm terminates in a local optimum [Jain et al, 1999]. This operator takes a solution as input and complements a randomly selected bit. For example $[001000]$ is produced by mutating the third bit of the input string $[000000]$. Mutation is used to increase the probability that the search space is sufficiently explored.

[Jain et al, 1999] report that *"the sensitivity of GAs to the selection of their parameters such as the population size, crossover and mutation probabilities is a major problem"*. Researchers defined problem-specific heuristics to alleviate this problem.

6.1.6 Data abstraction

In many applications the clusters the grouping produces must be represented in a compact form, simply because of their huge size. The entities that form the clusters are abstracted from to achieve this. [Jain et al, 1999] describe several representation schemes:

- **Representative elements** represent clusters by their centroid or a set of distant points. Distant points of a cluster are elements located at its edges.
- **Classification tree** represent the clusters by a classification tree that graphically visualises the search space and the cluster boundaries.
- **Logical expressions** represent the clusters by predicates that hold for all elements in the cluster, for example $y_1<4 \wedge y_2>3$, where y_1 and y_2 are two numeric features.

In practice, clusters usually are represented by their centroid [Jain et al, 1999].

6.1.7 Assessment of output

Assessment of output is an optional task that consists of an, often subjective, validation of the grouping result. Three types of assessment can be distinguished, all using statistical measures [Jain et al, 1999]:

- An **external assessment** compares the grouping to an a priory structure.
- An **internal examination** considers if the grouping is intrinsically appropriate for the data.
- A **relative test** compares two different groupings.

6.2 Clustering-based architecture reconstruction

This paragraph discusses several clustering-based architecture reconstruction approaches reported in literature. This overview is based on [Wiggerts, 1997], [Tzerpos and Holt, 1998], [Koschke, 2000] and [Mitchell, 2002].

6.2.1 Early architecture clustering

[Schwanke, 1991] describes a tool called Arch, which is a graphical and textual structure-chart editor for understanding and reorganising the internal structure of software systems.
Arch provides a semi-automatic architectural-clustering method that clusters procedures into modules. This is implemented with a hierarchical agglomerative clustering algorithm that uses the single linkage update rule. The similarity measure is based on shared design decisions; procedures are related if they share design decisions. Examples are procedures that use the same tables or call the same procedures.
The clustering can be used in three ways:

- **Batch clustering** runs without supervision.
- **Interactive radical clustering** asks the user for confirmation each time two clusters are combined.
- **Interactive reclustering** uses a previous clustering to guide the clustering.

An interesting feature of Arch is “maverick analysis”, which finds procedures that appear to be assigned to the wrong module. These are prioritised, and assigned to more appropriate modules.

[Choi and Scacchi, 1990] describe a fully automatic architectural clustering method that produces a hierarchical decomposition of a system. The method uses the NuMIL module interconnection language. In NuMIL a system is composed of subsystems, which are in turn composed of modules and other subsystems. This creates a hierarchy of subsystems and modules. In this hierarchy the subsystems correspond to interior nodes and the modules to leaf nodes. A module can be a single procedure or a set of procedures that are defined in a single source file.
The subsystem construction algorithm aims to minimize coupling and alteration distance. Coupling is a measure for the strength of the association between modules. The coupling of a system or subsystem is the sum of the couplings of all contained modules and subsystems. The *alteration distance* between modules is a measure for the distance between an altered module and the affected module. If both modules are located in the same subsystem the alteration distance is zero. Otherwise, the alteration distance is the length of the path between the altered and the affected module. The alteration distance for a system or subsystem is the sum of the alteration distances of the contained modules or subsystems.
The clustering algorithm starts with a resource flow diagram (RFD), in which the nodes represent the modules. An edge is placed from module A to module B if and only if module A provides one or more resources to module B. The clustering algorithm searches articulation points in the RFD, which are nodes that divide the RFD graph into two or more connected components. Together with the subgraphs, the articulation points become subsystems. When all articulation points have been processed the algorithm cleans the resulting hierarchy by removing subsystems with a single node and placing their content in a higher-level subsystem.

6.2.2 Rigi

Rigi [Rigi, 2004] is an architecture reconstruction tool (see paragraph 3.3.3). Rigi’s clustering method [Müller and Uhl, 1990] is intended to assist users with the reconstruction of the architecture of procedural software. Alternative decompositions are generated to achieve this.

It is chosen not to construct the decompositions fully automatically because *"an experienced software engineer will always be able to produce a better system decomposition than an automatic procedure –given sufficient time"* [Müller and Uhl, 1990].

Rigi produces a hierarchical decomposition in the form of a (k,2)-partite graph. Such a graph consists of a series of graph levels (or layers) and a special set of edges. These layers are connected by so called *layer-edges*, which may only connect adjacent layers. The nodes within each layer are connected by at most k edges. In terms of the software architecture, the nodes in the graph represent system entities, such as subsystems, modules or files. The levels are resource-flow graphs, each representing a certain abstraction level. The lowest level consists of source-code entities. Nodes at higher levels are composed of lower level nodes.

The clustering process consists of five steps [Müller and Uhl, 1990], which are described below. Users can invoke each of these steps from the Rigi application.

1. **Remove omnipresent nodes**. Omnipresent nodes are nodes that use many other nodes, or are themselves used by many other nodes. The first case represents *driver nodes*, and the second *library nodes*. Omnipresent nodes (and all their edges) are removed before the actual clustering takes place because they obscure the system structure.
2. **Compose by standard library** places known library members into special subsystems.
3. **Compose by interconnection strength** is a step that is based on the principles of high cohesion and low coupling. The interconnection strength similarity measure of two nodes in a resource flow graph is defined as *"the exact number of syntactic objects exchanged between the two nodes"* [Müller and Uhl, 1990]. Two nodes are *strongly coupled* if and only if their interconnection strength exceeds a certain threshold. Two nodes are *loosely coupled* if and only if their interconnection strength is below some other (lower) threshold. This step places strongly coupled nodes in the same subsystem and loosely coupled nodes in different subsystems. Node pairs that fall in neither category are placed in the same subsystem.
4. **Compose by common neighbour** is based on the software engineering principle of few interfaces. The intention is to identify pairs of loosely coupled nodes that have common clients or common suppliers. Placing such nodes into the same subsystem reduces the number of interfaces.
5. **Clean-up layers** identifies subsystems that contain only one node. Such subsystems are merged with their parent nodes.

6.2.3 ACDC

The Algorithm for Comprehension-Driven Clustering (ACDC) [Tzerpos and Holt, 2000] combines pattern detection and clustering techniques.

Instead of focussing on high cohesion and low coupling, ACDC uses a different approach to reconstruct an architecture. It is based on three characteristics that are considered essential for a recovered architecture to be useful for program understanding [Tzerpos and Holt, 2000]:

- **Effective cluster naming**: refers to the "data abstraction" issue discussed in paragraph 6.1.6. Giving meaningful names to the clusters that are familiar to maintainers makes an architecture much easier to understand, as opposed to names like "SS0", "SS1" et cetera.
- **Bounded cluster cardinality**: Clusters containing a lot of entities are not considered useful because of the overwhelming amount of information they present to the user. Clusters containing one or two entities are not considered useful either. Ideally, the clusters should contain between about five and twenty entities.
- **Pattern-driven**: A structure is easier to understand if it is presented in the form of familiar patterns, as was described in paragraph 2.4.3.

The ACDC algorithm works on procedural source code and clusters "source code resources". Examples of such resources are source-files, procedures, functions and variables. The algorithm reconstructs the architecture in two steps. In the first step, pattern-based techniques are used to detect common subsystem patterns. This produces a skeleton architecture. In the second step, orphan adoption techniques are used to classify entities that were not classified in the first step.

The first step of the algorithm performs the following substeps [Tzerpos and Holt, 2000]:

1. **Source file clusters** group entities that are defined in the same file into a cluster. This is only applicable if procedures, functions or variables are clustered.
2. **Body-header conglomeration clusters** group files that contain the definition and implementation of the same entities. In the C programming language for example .h files contain declarations and .c files the corresponding implementations.
3. **Leaf collection and supporting library identification** identifies sets of files that have a very large fan-in. These are potential library modules, but are not placed in a separate cluster yet.
4. **Ordered and limited subgraph domination** is the main step of the algorithm. A call graph is considered to find domination-relations between entities. If an entity x dominates an entity y this means that all paths in the call graph leading to y also contain x. When domination relations are found, the dominating and dominated entities are placed in the same cluster. This step ignores entities with a very large fan-out, as these are likely to be central entities that perform high-level control functions.
5. **Creation of support subsystem** is the last substep of the skeleton construction. In this step any entities found in step 3 that where not added to some cluster in step 4 are added to a special subsystem containing libraries.

In the second step, the ACDC algorithm uses the orphan adoption technique [Tzerpos and Holt, 1997] to place any remaining unclassified entities into the most appropriate cluster. Orphan adoption uses structural information to determine the set of entities to which some entity is closest related. This entity is then added to the cluster that contains the entities with which it has the highest number of relations. For further details the interested reader is referred to [Tzerpos and Holt, 1997].

[Tzerpos and Holt, 2000] describe two case studies that apply the ACDC algorithm to reconstruct an architecture from procedural source code. The first analyses Tobey, an industrial system consisting of 939 source files (250 KLOC). The second analyses Linux, an open source operating system. The analysed version consisted of 955 source files (750 KLOC).
The clusterings ACDC produced are compared to authoritive decompositions produced by experts on the analysed systems with the MoJo metric, which is discussed in paragraph 6.3.2. [Tzerpos and Holt, 2000] conclude that the ACDC algorithm produces meaningful decompositions that *"are among the better ones an automatic clustering algorithm can achieve"*.

6.2.4 Bunch

[Mitchell, 2002] describes a clustering tool called Bunch that implements three different partitional clustering algorithms [Bunch, 2005]. Bunch obtains hierarchical clusterings by repeated application of the algorithm. This is explained later in this paragraph.

Bunch is based on a module dependency graph $G=(V,E)$, in which V represents a set of entities and E a bag of directed, weighted edges between the entities. E can contain multiple edges between the same entities, possibly with the same weight. Bunch uses third party fact extractors to extract G from the source code.

Bunch is developed to cluster procedural software, but can also be used for object-oriented software. [Mitchell, 2002] describes several case studies where Bunch is used for procedural code. In those the module dependency graph $G=(V,E)$ is used with:

- E: set of source files.
- V: set of relations between the files in E. More specific, type references, variable access, function calls and macro invocations are considered. All relationship types have the same weight.

For cases where Bunch is used to cluster object-oriented source code [Mitchell, 2002] suggests to use the module dependency graph $G=(V,E)$ with:

- E: set of classes.
- V: set of relations between the classes in E, such as inheritance and associations. All relationship types have the same weight.

The similarity measure implemented in Bunch is called modularisation quality (MQ), and is based on the notions of intra- and inter-connectivity. Intra-connectivity measures the degree of connectivity within a cluster. Inter-connectivity measures the degree of connectivity between two clusters. Both are bounded by zero and one. Let N_i be the number of entities in a cluster i, and μ_i the total number of edges between all pairs of different entities in cluster i. Further, let $\varepsilon_{i,j}$ be the total number of edges between all pairs of entities of which one entity is located in cluster i and the other in j or vice versa. Then the intra-connectivity A_i of a cluster i and the inter-connectivity $E_{i,j}$ between clusters i and j are defined as [Mitchell, 2002]:

$$A_i = \frac{\mu_i}{N_i^2} \qquad E_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \dfrac{\varepsilon_{i,j}}{2N_iN_j} & \text{if } i \neq j \end{cases} \tag{23}$$

$A_i=0$ indicates that there are no dependencies within cluster i and $A_i=1$ that every entity in cluster i depends on all other entities in this cluster. $E_{i,j}=0$ indicates that none of the entities in cluster i depends on an entity in cluster j and vice versa. $E_{i,j}=1$ indicates that every entity in cluster i depends on every entity in cluster j and vice versa.

Figure 23 shows an example clustering of five entities $x_1,\ldots,x_5$ (the circles), in two subsystems (the squares). Subsystem 1 has one intra-edge and contains two entities, so $A_1={}^1/_4$. Two inter-edges between subsystem 1 and 2 are present, giving $\varepsilon_{1,2}=2$. Because $N_1=2$ and $N_2=3$ $E_{1,2}={}^2/_{(2\times2\times3)}={}^1/_6$.

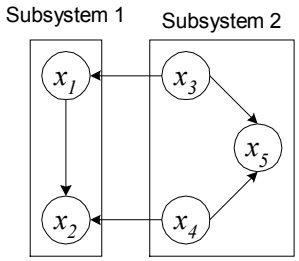


Figure 23: MQ calculation example

If k is the number of clusters and (23) defines intra- and inter-connectivity, the BasicMQ similarity metric is defined as (adapted from [Mitchell, 2002]):

$$BasicMQ = \begin{cases} \dfrac{1}{k}\sum_{i=1}^{k} A_i - \dfrac{\frac{1}{k(k-1)}}{2} \sum_{i,j=1 \wedge i \leq j}^{k} E_{i,j} & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{cases} \tag{24}$$

In (24), for the case $k>1$ the left summation-term calculates the average cohesion and the right one the average coupling. The metric rewards cohesion and penalizes coupling. The BasicMQ metric is bounded by –1 (no intra-edges) and +1 (no inter-edges) [Mitchell, 2002]. Mitchell defines the BasicMQ similarity metric slightly differently, namely without the $i \leq j$ term. However, the metric is only bounded by –1 if this term is added. Further, the examples in [Mitchell, 2002] and [Mitchell et al, 1998] also utilize this addition.
For the example in Figure 23

$$BasicMQ = \frac{1}{2}\sum_{i=1}^{2} A_i - \frac{\frac{1}{2\times1}}{2} \sum_{i,j=1 \wedge i \leq j}^{2} E_{i,j} = \tfrac{1}{2}\times\left(\tfrac{1}{4}+\tfrac{2}{9}\right) - \tfrac{1}{4}\times\left(\tfrac{1}{6}\right) = 0{,}1944$$

Worst case $O(|E|)=O(|V|^2)$, giving the BasicMQ calculation a worst-case computational complexity of $O(|V|^4)$. Because in practice $O(|E|)\approx O(|V|)$ the average complexity is $O(|V|^3)$ [Mitchell, 2002]. This is still too expensive for large graphs. Another disadvantage of this metric is that it does not allow the edges to have different weights.

The TurboMQ metric solves both problems. Let k represent the number of clusters again, and μ_i the *summed weight* of the intra-edges within cluster i. Further, let $\varepsilon_{i,j}$ and $\varepsilon_{j,i}$ represent the *summed weights* of the inter-edges from cluster i to j and vice versa respectively. If $i=j$ then $\varepsilon_{i,j}=\varepsilon_{j,i}=0$. Using these definitions TurboMQ is defined as [Mitchell, 2002]:

$$TurboMQ = \sum_{i=1}^{k} CF_i$$

$$CF_i = \begin{cases} 0 & \text{if } \mu_i=0 \\ \dfrac{\mu_i}{\mu_i + \dfrac{1}{2}\displaystyle\sum_{j=1 \wedge j\neq i}^{k} \left(\varepsilon_{i,j} + \varepsilon_{j,i}\right)} & \text{if } \mu_i \neq 0 \end{cases} \qquad (25)$$

CF_i is called the *cluster factor* of cluster i.

Bunch implements three different partitional clustering algorithms [Mitchell, 2002]:

- An **exhaustive algorithm** that enumerates all possible partitions and returns the partition with the highest MQ value. The computational complexity of this algorithm is $O(|V|!)$, so it can only be used for very small module dependency graphs.
- A **hill-climbing algorithm** that is based on the k-means algorithm discussed in paragraph 6.1.5 (“Square error clustering algorithms” section). To reduce the risk of finding a local optimum it does not generate one initial partition like the k-means algorithm, but a set of initial partitions. All partitions in this set are clustered, after which the best result is chosen. The computational complexity of the hill-climbing algorithm is $O(|V|^2\cdot|E|)$.
- A **genetic algorithm**, similar to the genetic algorithm discussed in paragraph 6.1.5 (“Evolutionary clustering algorithms” section). The algorithm described in that section uses a binary encoding, giving a partition with two clusters. Both [Mitchell, 2002] and [Doval et al, 1999] give an example with a quartairy encoding (i.e. four clusters), but the chosen number of clusters is not mentioned explicitly. In Bunch’ genetic algorithm, parameters like the population size and number of created generations depend on the number of entities. If N is the number of entities (so $N=|V|$), the genetic algorithm uses a population size of $10N$, and creates $200N$ generations before terminating. These figures were determined empirically. A version of the MQ metric is used as fitness function.

[Mitchell, 2002] compares the results of the genetic algorithm to that of the hill-climbing algorithm. This shows that the quality of the results of the genetic algorithm varies more than that of the hill-climbing algorithm. The execution time of the genetic algorithm also varies widely. [Mitchell, 2002] speculates that the chosen encoding causes this, and concludes that the genetic algorithm must be improved further in order to be useful.

[Shokoufandeh et al, 2004] apply spectral methods to the software-clustering problem. The resulting algorithm is called the recursive bisection algorithm. The quality of clusterings is frequently evaluated with the notions of conductance volume and normalised inter-cluster distance [Kannan et al, 2004]. [Shokoufandeh et al, 2004] show that their algorithm has a conductance volume and an inter-cluster volume within a known distance of the optimal clustering. This means that their algorithm gives a bounded approximation of the optimal clustering. For more details on conductance volume and an inter-cluster volume the interested reader is referred to [Kannan et al, 2004].
A disadvantage of the recursive bisection algorithm is that it has a relatively large worst-case complexity. If N is the number of entities, it is $O(N^4)$. [Shokoufandeh et al, 2004] applied their

algorithm and the hill-climbing algorithm of Bunch[21] to cluster 13 software systems and compared the results. They observed that their algorithm *"is generally worse than Bunch in quality of solutions and running time, and only gets worse as the size of the input increases"*. This implies that the hill-climbing algorithm of Bunch efficiently yields clusterings within a bounded approximation of the optimal solution.

Although partitional algorithms are used, Bunch can still produce hierarchical decompositions. This is done by repeatedly applying the clustering algorithm to the result of the previous clustering [Mitchell, 2002]. The first clustering cycle clusters the module dependency graph the user provided. Say this produces a clustering K_1. Based on K_1 a new module dependency graph is created that will be clustered again. Say this graph is $G_2=(V_2,E_2)$, where V_2 represents the entity-set and E_2 the set of edges between the entities. For each cluster in K_1 a node is added to V_2. The set of edges E_2 is defined by considering the nodes in the clusters of K_1. For every pair of entities $v,w \in V_2$ for which the corresponding clusters in K_1 contain nodes with an edge between them, an edge is added to E_2. The weight of this edge is the sum of the weights of the edges between the clusters represented by v and w.
After the new module dependency graph G_2 has been constructed, it is clustered. This process of constructing a new module dependency graph and clustering it is repeated until a clustering is obtained with only one cluster.

[Mancoridis et al, 1999] present several extensions to Bunch that were added in response to user-feedback:

- **Library and omnipresent-module detection** allows the identification of modules that are used everywhere in the software. The user can do this manually, or it can be done automatically by considering the fan-in and fan-out of the modules, as [Müller et al, 1993] described. Modules with a fan-out that exceeds the average value three times are considered omnipresent modules and are ignored during the clustering process. Instead, they are placed in a special subsystem. Modules with a fan-in that exceeds the average value three times are considered library modules, and are treated similarly.
- **User-directed clustering** allows users to define clusters manually to incorporate their knowledge in the clustering process.
- **Orphan adoption** is used to allow incremental clustering, as [Tzerpos and Holt, 1997] described. An orphan is a module that is either completely new or has undergone structural changes that might justify placement in a different subsystem. The hill-climbing and genetic algorithms of Bunch are non-deterministic. If a clustering has been generated earlier and new entities need to be incorporated, constructing a completely new clustering can lead to a completely different result. Orphan adoption adds orphans to existing clusters, instead of constructing a completely new clustering. This enables users to preserve an existing subsystem structure.

Bunch uses dotty [North and Koutsofios, 1994] to visualise the resulting clustering, but it is also possible to export the clustering to a file for integration with other visualisation tools.
[Mitchell, 2002] mentions the use of the Acacia fact extractors (see paragraph 3.4.2) for C code and the Chava fact extractor [Korn et al, 1999] for Java code. However, the simple format of the module dependency graph enables easy integration of other fact extractors.

Besides by means of the in- and output-files, Bunch can also be integrated into reverse engineering frameworks by means of the Bunch API [Mitchell, 2002]. This API offers a Java [Java, 2005] interface to the internal components of Bunch.

Bunch is used in various reverse engineering case studies. As described above, [Shokoufandeh et al, 2004] empirically compared the clusterings produced by Bunch to those produced by an algorithm whose results are within a known approximation of the global optimum.
[Mitchell, 2002] reports on the use of Bunch to recover the architecture of dotty and Bunch itself. The result of the clustering is compared manually to an expert's decomposition. In both case it is concluded that Bunch produces valid clusterings.

[21] [Shokoufandeh et al, 2004] used Bunch with the default settings.

[Tzerpos and Holt, 1999] use Bunch to evaluate an implementation of the MoJo similarity metric, which is discussed in paragraph 6.3.2.
[Mitchell and Mancoridis, 2002] describe experiments with Bunch to investigate the effect of several user-defined parameters. Five subject systems were analysed, with the number of entities varying between 13 and 413. The MQ metric is used to evaluate the results.

[Anquetil and Lethbridge, 1999] use Bunch to experiment with different clustering configurations. They experiment with various different feature choices and determine the quality of the clustering based on precision and recall, which are discussed in paragraph 6.3.1. Four large software systems, written in C and Pascal (140-2000 KLOC), are used as test subjects.
[Anquetil and Lethbridge, 1999] distinguish two types of features: formal and non-formal features. Formal features directly affect the behaviour of the software, for example type references, variable references and procedure calls. Non-formal features do not directly affect the behaviour of the software, for example frequently used words in identifiers or comments. Based on their experiments, [Anquetil and Lethbridge, 1999] conclude that non-formal features can also produce good clusterings. These features have the advantage over formal features that they offer less redundancy.

6.2.5 Alborz

[Sartipi and Kontogiannis, 2002] describe a supervised clustering technique for procedural software. The technique represents the analysed system with an attributed relational graph in which the nodes represent source code entities, like for example source-files, functions, data types and global variables. The edges in this graph represent relationships between the entities, like for example calls, defines, updates and declares.

The clustering algorithm uses the “maximal association” coefficient, which calculates the similarity of two entities based on the maximum number of shared features. Data mining techniques are used to extract the values for the coefficient from the attributed relational graph.

The clustering algorithm is based on the k-means algorithm, which is described in the “Square error clustering algorithms” section in paragraph 6.1.5. The algorithm is designed to be able to handle large search spaces. This is achieved by dividing the clustering space into a number of smaller, user-selected subspaces, and iteratively processing these subspaces. The impact of incorrect placements is reduced in two ways: First of all, later iterations can reassign entities that were assigned to some cluster in earlier iterations. Second, at the end of each iteration the user can manually correct misplacements.

The describe approach has been implemented in a tool called Alborz. This tool has been applied to six industrial software systems, (35 to 74 KLOC). The clustering results are evaluated with the precision and recall metrics, which are described in paragraph 6.3.1. Precision varied between 43% and 94%, and recall between 33% and 100%. These results are considered satisfactory [Sartipi and Kontogiannis, 2002].

6.2.6 LIMBO

[Andritsos and Tzerpos, 2003] use the LIMBO algorithm to reconstruct the architecture of procedural source code.

The LIMBO algorithm (scaLable InforMation Bottleneck) is an agglomerative hierarchical algorithm that is based on information loss minimization. The algorithm uses a feature matrix as input. In this matrix the rows denote entities, in this case source files, and the columns boolean features, for example developer-names, directory paths or dependencies to source files. This allows the combination of structural and non-structural features.
As all agglomerative hierarchical algorithms, LIMBO starts with an initial clustering in which every entity is converted into a cluster. Based on the entropy of the feature matrix, clusters are combined until a predefined number of clusters remains. The algorithm combines the pair of clusters that gives the smallest reduction of the uncertainty (entropy) of the feature matrix. Besides stopping when a certain number of clusters has been reached, the algorithm also

stops combining clusters when subsequent clusters only differ in the allocation of one entity or the merging/splitting of one cluster. This is determined with the MoJo metric[22], which is described in paragraph 6.3.2.

[Andritsos and Tzerpos, 2003] report the application of LIMBO to Linux (955 files, 750 KLOC) and Tobey (939 files, 250 KLOC). The MoJo metric is used to compare the result with that of several other clustering algorithms, including Bunch and ACDC. In this experiment all clustering algorithms used the same structural features. LIMBO's decompositions achieved a lower (=better) MoJo value than those produced by the other algorithms.
The addition of features based on the developer-names, source directory, lines of code (discretized) and development time (month and year) improved LIMBO's clusterings further.

6.2.7 InSight

Klocwork InSight [Klocwork, 2005] is a commercial architecture reconstruction and analysis tool. As described in paragraph 3.3.7, InSight identifies nine architectural views, including the code, package and component views. Some of these views are extracted from the source code, whereas the user defines the others. This paragraph describes how the latter is achieved

At the basis of InSight's code, package and component views lies the "summary model" [Mansurov and Campara, 2003]. This model essentially is an attributed graph $G=(N,E)$ where $N=N_r \cup N_s$ and $E=E_r \cup E_s$. N_r and E_r represent entities defined in the source code and relations specified in the source code between these entities respectively. The nodes in N_r have an attribute that stores a reference to the concerned location in the source file. N_s represents the set of summary nodes, which are user-defined aggregations of node sets. Finally, $E_s \subseteq N_s \times N$ represents the set of edges between summary nodes and the nodes they aggregate.

[Mansurov and Campara, 2003] informally describe three operations on the summary model:
- **Aggregation** defines a new summary node that aggregates a set of child nodes.
- **Detalization** is the inverse of aggregation. It removes a summary node and replaces it with its child nodes.
- **Trimming** moves a node to another location in the graph.

With these three operations users can manually reconstruct an architecture from source code. Users can also experiment with the summary model to ask "What if?" questions. These can be answered by applying envisioned changes, after which InSight visualises their impact on the architecture.

6.2.8 ACT

[Bauer and Trifu, 2004] describe an elaborate architectural-clustering method called "architecture aware, adaptive clustering". This method combines clustering and pattern detection to recover meaningful system decompositions. The method represents the static structure of the software as a graph in which nodes represent classes and weighed edges relations between these classes. The weights represent semantic information about classes and are determined with "architectural clues". Three types of architectural clues are identified, namely method types, library classes and design patterns.

Prolog rules are used to detect instances of seven structural design patterns in the source code. The detected patterns are Template method, Abstract factory, Strategy, Composite, Proxy, Adapter and Façade [Gamma et al, 1995].

[22] The MoJo similarity metric is defined as the minimal number of move- and join-operations required to transform one clustering into the other or vice versa. This metric is described in paragraph 6.3.

The method-types classify methods according to three criteria:

- The **kind** of a method describes its function. Examples are constructor, empty, accessors, template, factory and normal.
- The **inheritance statue** of a method describes the role of the method in an inheritance tree, if inheritance is present at all. Possible values are implementing, extending, overriding, adding or new.
- The **usage** criterion classifies methods according to their usage into initialisation, public interface, protected interface and implementation methods.

The third type of architectural clue, library classes, is used to detect library modules. Based on the number of classes that call a suspected library class, a library recogniser module detects library classes. This is similar to the method [Tzerpos and Holt, 1997] described.

The architectural clues are used to determine the coupling between the classes from six points of view [Trifu, 2003]:

- **Inheritance coupling** takes the different contexts of inheritance relations into account. This is necessary because inheritance can be used for many different purposes, including specialisation and implementation reuse.
- **Aggregation coupling** distinguishes composition from the other aggregation types.
- **Association coupling** refers to coupling through method parameters, method return types and local variables.
- **Access coupling** determines the coupling through access of class attributes.
- **Call coupling** determines the degree of coupling by the number of method calls between the classes.
- **Indirect coupling** is coupling through common usage of a resource. The assumption that this type of coupling relates classes is based on the software engineering principle of few interfaces; grouping classes that use the same resource reduces the number of subsystems that depend on the subsystem with the used class [Müller and Uhl, 1990].

For details on how these types of coupling are related to the architectural clues the interested reader is referred to [Trifu, 2003].

The weights of the edges between the classes are based on the weighted sum of the values for the six types of coupling. The weight of each type is based on personal experience and intuition [Trifu, 2003]. The resulting graph is clustered with the “modified-MST” algorithm. This algorithm is an improved version of the MST algorithm discussed in paragraph 6.1.5 (“Graph-theoretic clustering algorithms” section), which uses a different heuristic to obtain the clusters.

[Trifu, 2003] describes the implementation of architecture aware, adaptive clustering in a framework called Adaptive Clustering Testbed (ACT). ACT is used to compare the results obtained with the described method to a conventional clustering technique. This technique does not use the architectural clues and does not consider indirect coupling [Trifu, 2003]. The resulting clusterings are assessed using both the MoJo metric (see paragraph 6.3.2) and the average cohesion and coupling of the clusters.

Two case studies were performed to compare architecture aware, adaptive clustering to the conventional clustering method, namely using the Java AWT library (482 classes, 142 KLOC) and the SSHTools project (507 classes, 76 KLOC) as input. The results show that architecture aware, adaptive clustering produces more accurate clusterings than the conventional clustering technique.

6.3 Clustering result evaluation

This paragraph discusses several methods to assess the output of the clustering process. The selection is based on [Wen and Tzerpos, 2004a] and [Mitchell, 2002].

In general clusterings are evaluated with an external or internal assessment, or a relative test, as is described in paragraph 6.1.7. These methods can be applied to architectural clusterings as follows:

1. **External assessment:** In a manual inspection it is checked if the proposed clustering *seems* appropriate. Often the developers of the system are consulted. However, this method is highly subjective.
2. **Internal assessment** uses metrics that only consider the proposed clustering itself. This evaluation method takes the goal of architectural clustering, program comprehension, into account indirectly through the choice of the metric.
3. **A relative test** compares the clustering result to an a priori structure. This is considered the ideal assessment method to evaluate the quality of architectural clusterings [Mitchell, 2002], [Koschke, 2000]. However, it has the disadvantage that an expert's decomposition must be available to which the clustering result can be compared.

The first and last methods are most frequently used in architectural clustering literature. This paragraph discusses several evaluation methods that fall in the last category. These methods compare the similarity of the produced clustering to an expert's decomposition using some similarity measure. Note that these similarity measures are not the same as the similarity measures used during the clustering process (which are described in paragraph 6.1.4). Those quantify the similarity of two *entities*, whereas the measures discussed here quantify the similarity of two *clusterings*.

6.3.1 Precision and recall

[Anquetil and Lethbridge, 1999] use precision and recall to compare a clustering result to an a priori structure created by experts on the analysed systems. The latter is called an *expert decomposition*.

In a clustering any pair of entities can either be placed in the same cluster or in a different one. In the first case this is called an *intra pair* and in the second an *inter pair*. Let K and D be two partitionings of the same entities such that K is produced by an architectural clustering method and D is an expert decomposition. Then *precision* is defined as the percentage of intra pairs in K that are also intra pairs in D. *Recall* is defined as the percentage of intra pairs in D that are also intra pairs in K.

Figure 24 shows two partitionings labelled K and D of the entity-set $\{x_1, \ldots, x_6\}$. Partitioning K consists of the clusters K_1 and K_2. Partitioning D consists of the clusters D_1 and D_2. Observe that the only difference between K and D is the placement of x_4. In this example K_1 contains six intra-pairs, K_2 one, and D_1 and D_2 both three. Of the seven intra-pairs in partition K, D contains four, so precision=$4/7=57\%$. Of the six intra-pairs in D, K contains four, so recall=$4/6=67\%$.

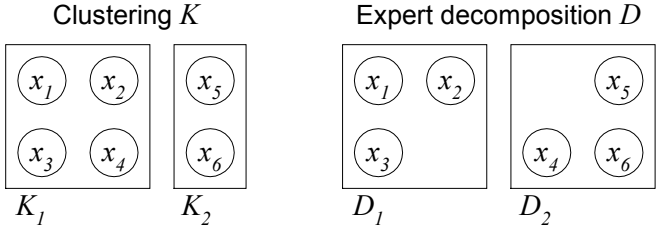


Figure 24: Precision/recall example

Though frequently used to evaluate clustering results, precision/recall has several limitations [Mitchell and Mancoridis, 2001]. First of all, the calculation does not consider edges. Every incorrect placement has the same weight. In literature incorrect placements that affect many edges are usually considered more important than those that affect little edges, which suggests that the calculation should also consider the edges. Second, the measurement is sensitive to the number and size of the clusters. A few misplaced modules in a cluster with

relatively few members have much more impact on precision/recall than when the cluster has many members. Finally, the number and size of the clusters impacts precision/recall.

[Koschke, 2000] presents a framework for the evaluation of clustering algorithms for architecture reconstruction. This framework compares the produced decompositions against expert decompositions with a metric that is based on precision and recall.

6.3.2 MoJoQuality, EdgeMoJo and MoJoFM

MoJoQuality

[Tzerpos and Holt, 1999] define the MoJo metric for the comparison of two decompositions. It the distance between two decompositions in terms of the minimal number of move and join operations that is required to transform one decomposition into the other. A move operation relocates a single entity from one cluster to another cluster. A join operation merges two clusters. Let K and D be two decompositions of a system consisting of N entities and let $mno(K,D)$ be the minimum number of move and join operations to transform K into D. If $x \downarrow y$ refers to the minimum of x and y, $MoJo(K,D)$ is defined as [Tzerpos and Holt, 1999]:

$$MoJo(K,D) = mno(K,D) \downarrow mno(D,K) \tag{26}$$

[Wen and Tzerpos, 2003] describe an efficient algorithm to compute the MoJo distance between two decompositions, K and D. The total computational complexity of this algorithm is $O(N \cdot log\ N + (L+M) \cdot L \cdot M)$, where L and M are the number of clusters in K and D respectively.

Figure 25 shows two partitionings labelled K and D of the entity-set $\{x_1,...,x_6\}$. K and D consist of the clusters K_1 and K_2, and D_1 and D_2 respectively. Observe that the only difference between K and D is the placement of x_2 and x_4. The minimum number of move and join operations to transform K into D or vice versa is two, so $MoJo(K,D)=2$.

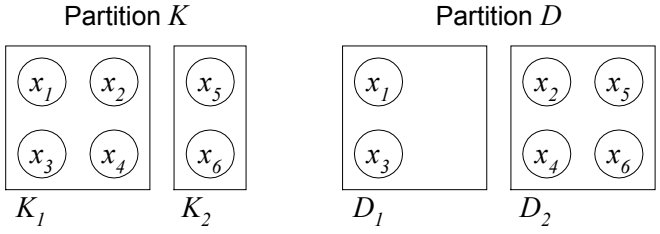


Figure 25: MoJo Example

If K refers to a decomposition produced by a clustering algorithm and D to an expert decomposition, [Tzerpos and Holt, 1999] define the quality of K relative to D as

$$MoJoQuality(K,D) = \left(1 - \frac{MoJo(K,D)}{N}\right) \times 100\% \tag{27}$$

$MoJoQuality(K,D)=100\%$ indicates that the clustering is the same as the expert decomposition. Because any partition of N entities can be transformed into any other partition using N moves [Tzerpos and Holt, 1999], $MoJo(K,D) \leq N$. Hence $MoJoQuality(K,D) \geq 0\%$. Suppose that in the example of Figure 25 K is a decomposition produced by a clustering algorithm, and D the expert decomposition. Because six entities are clustered, $MoJoQuality(K,D)=(1-{}^2/_6) \times 100\%=67\%$.

[Tzerpos and Holt, 2000] describe two case studies where the MoJoQuality metric is used to evaluate the quality of decompositions produced by the ACDC algorithm. The two decompositions achieve a MoJoQuality of 56% and 64%. This is claimed to be among *"the higher ones an automatic clustering algorithm can hope to achieve"* [Tzerpos and Holt, 2000].

This matches with the results reported by [Trifu, 2003]. Relative to the existing package structure of two software systems, the method described by [Trifu, 2003] achieves a MoJoQuality of about 50% to 65%.

EdgeMoJo

A disadvantage of the MoJo metric is that it does not take edges into account. Suppose that a clustering algorithm is used to decompose a system consisting of N entities twice, producing decompositions K_1 and K_2, and that D is an expert decomposition of the analysed system. Further suppose that K_1 and K_2 both have one entity, say x_1 and x_2 respectively $(x_1 \neq x_2)$, that is placed in a different cluster in D, and that x_1 has one edge to other entities and x_2 has ten. Then $MoJo(K_1,D)=MoJo(K_2,D)$, indicating that K_1 and K_2 are equally good. But because the misplacement of x_2 is clearly more important than that of x_1, K_1 is actually better than K_2.

[Wen and Tzerpos, 2004a] describe an extended version of MoJo, called EdgeMoJo, which takes the number and weight of edges into account. This metric first calculates the MoJo value, after which the additional cost imposed by the edges is calculated. [Wen and Tzerpos, 2003] show that the order in which the MoJo metric performs the move and join operations is not relevant. Therefore, the EdgeMoJo algorithm starts with the join operations, after which the move operations are performed.
Let K_{new} denotes the cluster an entity x is moved to, K_{old} the cluster x used to be placed in, $|z|$ the absolute value of z and $W(x,K_i)$ the summed weight of the edges between x and the entities in cluster K_i. In the EdgeMoJo metric each move operation of entity x has the weight $m(x)$, instead of one as in the MoJo metric, with:

$$m(x) = 1 + \frac{\left|W(x,K_{new}) - W(x,K_{old})\right|}{W(x,K_{new}) + W(x,K_{old})} \tag{28}$$

Figure 26 shows the same partitionings as Figure 25 but now with edges between the entities. All edges have a weight of one. Recall that the only difference between K and D is the placement of x_2 and x_4 and that $MoJo(K,D)=2$. The aggregate weight between x_2 and the entities in K_1 $W(x_2,K_1)=2$. Similarly, $W(x_4,K_1)=1$, $W(x_2,D_2)=3$ and $W(x_4,D_2)=2$. Therefore the cost of relocating x_2 is $m(x_2)=1+{}^1/_5=1,2$. Similarly, $m(x_4)=1+{}^1/_3=1,3$. This gives $EdgeMoJo(K,D)=2,5$.

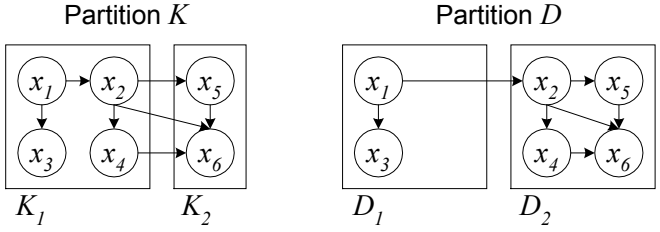


Figure 26: EdgeMoJo example

MoJoFM

[Wen and Tzerpos, 2004b] describe an improved version of the MoJo metric, called MoJoFM. This metric solves some anomalies of the MoJo metric, such as the tendency of MoJo to consider clusterings with singleton clusters[23] very good. However, MoJoFM does not take the edges into account.

MoJo for hierarchical decompositions

The metrics discussed so far compare two partitionings, so two clusterings without hierarchy. [Shtern and Tzerpos, 2004] informally describe a method to compare hierarchical

[23] Clusters containing one entity.

decompositions. Essentially the method converts both hierarchical decompositions into a set of partitionings and applies an existing similarity metric for partitionings to them. Next, the obtained set of metrics is aggregated to produce a single value that indicates the similarity of the hierarchical decompositions.

A hierarchical decomposition K is converted into a partitioning as follows. Suppose that a tree in which the nodes represent clusters and entities represents a hierarchical decomposition. Further, suppose that the edges of this tree represent containment relations between clusters and entities. Let the *level* of a cluster in K be the number of edges between the cluster and the root of the tree representing K. Further, let $height(K)$ be the depth of the tree representing K.
For each level l of K $(1 \leq l < height(K))$ a partioning is constructed as follows (adapted from [Shtern and Tzerpos, 2004]):

1. Assign each entity in each cluster with a level larger than l to its ancestor at level l. This produces a hierarchical decomposition K' such that of $height(K')=l+1$.
2. Create a new clustering K_l that contains the clusters that contain the leaves of K'. Now $height(K_l)=1$, so K_l is a partitioning of the classes.

The left side of Figure 27 shows an example of a hierarchical decomposition of four levels. This decomposition classifies the entity-set $\{x_1,...,x_9\}$ into the clusters A, A_1, A_2, A_3, A_4, A_5 and A_6. On the right side of the figure the result of step 1 and 2 of the above algorithm are shown for $l=2$. Observe that the classes originally placed in A_5 and A_6 are now placed in A_4.

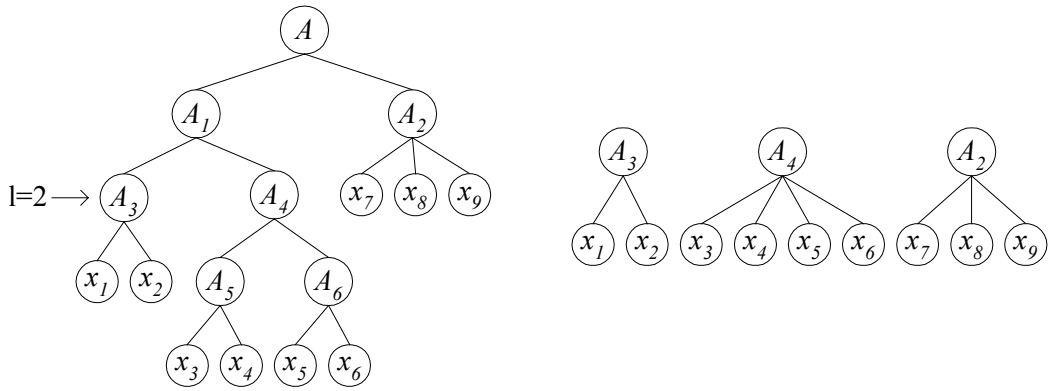


Figure 27: Example conversion of a hierarchical decomposition (left) into a partitioning (right) for level 2

Let K and M be two hierarchical decompositions. The similarity of K and M is computed as follows. First K and M are both converted to a set of partitionings using the method described above. Next, it is ensured that both of these sets have the same size. Let h_k and h_m be $height(K)$ and $height(M)$ respectively, and let K_i be the partitioning of K obtained with the above procedure for level i $(1 \leq i < h_k)$. If $h_k \neq h_m$, the decomposition with the lowest height is extended by copying its most detailed partition. For the case $h_k < h_m$ decomposition K_{h_k-1} is copied (h_m-h_k) times to obtain the partitionings numbered h_k up to h_m-1 of K. The case $h_k > h_m$ is treaded likewise with k and m exchanged. Suppose for example $h_k=3$ and $h_m=5$, as is shown in Figure 28. Then the partitionings for level 3 and 4 of K are copies of that for level 2.

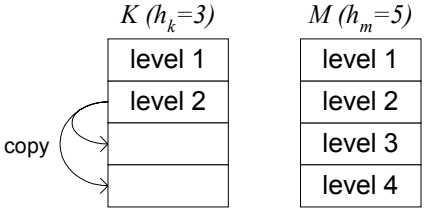


Figure 28: Example of level copying

To calculate the similarity of K and M, some similarity metric for partitionings, for example MoJo, is applied to every pair of decompositions at the same level. Let K_j and M_j be the partitionings of K and M respectively the previous algorithm produced for level j $(1 \leq j < (h_k \uparrow h_m))$[24]. Now the similarity is calculated for every level. Let S_j be the similarity of K_j and M_j, and w_j the weight of level j. The overall similarity sim is calculated with [Shtern and Tzerpos, 2004]:

$$sim = \sqrt{\sum_{j=1}^{h_k \uparrow h_m} \left(w_j \cdot S_j^{\,2} \right)} \qquad \text{with} \sum_{j=1}^{h_k \uparrow h_m} w_j = 1 \tag{29}$$

[Shtern and Tzerpos, 2004] use a linear weighting scheme that assigns equal weights to all levels, as is applied in (29).

6.3.3 EdgeSim and MeCl

The methods discussed in the previous paragraphs compare two decompositions based on the placement of entities in clusters. [Mitchell and Mancoridis, 2001] present two methods for comparing partitionings that are based on the relations between the entities. Both methods are based on the module dependency graph Bunch uses (see paragraph 6.2.4). In a module dependency graph $G=(V,E)$, V represents the set of entities (e.g. source files) to be clustered and E a bag of weighted edges between entities in V. If the entities represent source files, the edges in E represent dependencies between source files.

EdgeSim

Let A and B be two partitionings of G into l and m different clusters respectively. Further, let A contain the clusters $\{A_1, \ldots, A_l\}$ and B the clusters $\{B_1, \ldots, B_m\}$. [Mitchell and Mancoridis, 2001] distinguish two types of edges:

- Intra-edges that do not cross a cluster boundary.
- Inter-edges that do cross a cluster boundary.

The set Φ of intra-edges in both A and B, and the set Θ of inter-edges in both A and B are defined as (adapted from [Mitchell and Mancoridis, 2001]):

$$\begin{aligned} \Phi &= \left\{ (u,v,n) \in E \,\middle|\, 1 \leq i \leq l \wedge 1 \leq j \leq m \wedge \left(u \in A_i \wedge v \in A_i\right) \wedge \left(u \in B_j \wedge v \in B_j\right) \right\} \\ \Theta &= \left\{ (u,v,n) \in E \,\middle|\, 1 \leq i \leq l \wedge 1 \leq j \leq m \wedge \left(u \in A_i \wedge v \notin A_i\right) \wedge \left(u \in B_j \wedge v \notin B_j\right) \right\} \\ \Upsilon &= \Phi \cup \Theta \end{aligned} \tag{30}$$

The set Υ represents the set of edges that are *either* intra-edges *or* inter-edges in *both* A and B.

Let $weight(E)$ and $weight(\Upsilon)$ be the sum of the weights of the edges in E and Υ respectively (if an edge has no associated weight, a weight of one is assumed). Using these definitions the EdgeSim measurement is now calculated with ([Mitchell and Mancoridis, 2001]):

$$EdgeSim(A,B) = \frac{weight(\Upsilon)}{weight(E)} \times 100\% \tag{31}$$

Figure 29 shows two partitionings labelled A and B of the entity-set $\{x_1, \ldots, x_8\}$. A consists of the clusters A_1 and A_2, and B consists of the clusters B_1, B_2 and B_3. Observe that the difference between A and B is the placement of x_2, x_4, x_5 and x_6. The edges in the set Φ of intra-edges in both A and B are drawn in blue, and the edges in set Θ of inter-edges in both A

[24] $\uparrow$ gives the maximum of two numbers.

and B are drawn in red. The first contains 5 edges and the second 1. In total the module dependency graph contains 11 edges. Since the weight of all edges is one, this gives $EdgeSim(A,B)=6/11\times100\%=55\%$.

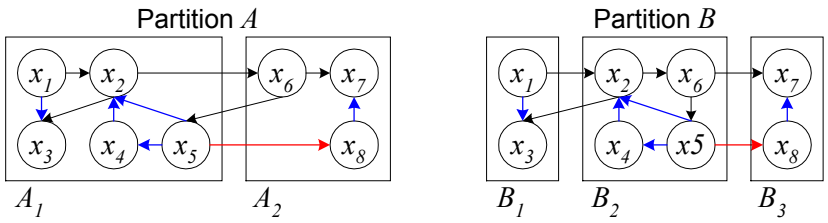


Figure 29: EdgeSim example

MeCl

The MeCl measure is also based on edge similarities. Again, let A and B be two partitionings of G into the clusters $\{A_1,\ldots,A_l\}$ and $\{B_1,\ldots,B_m\}$ respectively. Let $\Phi_A(A_i)$ denote the set of intra-edges in cluster A_i $(1\leq i\leq l)$ and $\Theta_B(B_j)$ the set of inter-edges connected to entities in B_j $(1\leq j\leq m)$. Then the set $\Upsilon_{i,j}$ of edges that are intra-edges in A_i and inter-edges connected to entities in B_j is defined as [Mitchell and Mancoridis, 2001]:

$$\Upsilon_{i,j}=\Phi_A\left(A_i\right)\cap\Theta_B\left(B_j\right) \tag{32}$$

Using (32) the set of edges that *became* intra-edges in B_j is defined as [Mitchell and Mancoridis, 2001]:

$$\Upsilon_{B_j}=\bigcup_{i=1}^{l}\Upsilon_{i,j} \tag{33}$$

If $weight(E)$ and $weight(\Upsilon_B)$ are the sum of the weights of the edges in E and Υ_B respectively[25], the MeCl measure is calculated with [Mitchell and Mancoridis, 2001]:

$$\begin{aligned}\Upsilon_B&=\bigcup_{j=1}^{m}\Upsilon_{B_j}\\ MeCl\left(A,B\right)&=\left(1-\frac{weight\left(\Upsilon_B\right)}{weight\left(E\right)}\right)\times100\%\end{aligned} \tag{34}$$

Figure 30 shows the same two partitionings as Figure 29. The set $\Upsilon_{1,1}$ of edges that are intra-edges in A_1 and inter-edges connected to entities in B_1 consists of the edge from x_1 to x_2, and of the edge from x_2 to x_3. These edges are drawn in blue. $\Upsilon_{1,2}$ equals $\Upsilon_{1,1}$. $\Upsilon_{1,3}$ and $\Upsilon_{2,1}$ are empty and $\Upsilon_{2,2}$ and $\Upsilon_{2,3}$ both consist of the red edge from x_6 to x_7. This gives $MeCl(A,B)=(1-{}^3/_{11})\times100\%=63\%$.

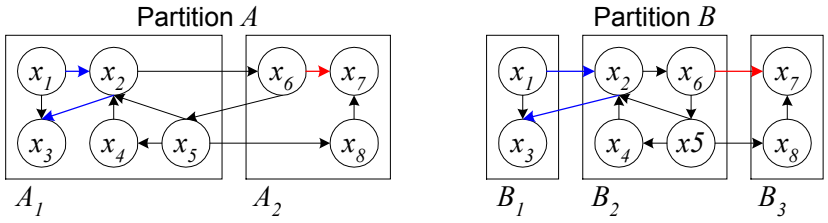


Figure 30: MeCl example

A disadvantage of both EdgeSim and MeCl is that the placement of entities in clusters is not considered at all [Wen and Tzerpos, 2004].

[25] Again, if an edge has no associated weight a weight of one is assumed

7 Case study: Architectural clustering

This chapter describes the second of the two case studies described in this thesis. This case study investigates the use of clustering-based architecture reconstruction methods, using the theory described in chapter 6. These methods automatically reconstruct a structural view of a software architecture from source code. In this chapter the terms "clustering-based architecture reconstruction" and "architectural clustering" are used interchangeably.

7.1 Case study goals

Clustering-based architecture reconstruction techniques use mathematical clustering techniques to reconstruct architectural components. These techniques find some natural grouping of data elements, in this case source code elements. A view of the system's as-built architecture is constructed by defining abstractions that group source code elements.

Chapter 6 describes several methods for architectural clustering reported in literature. Although it is known that it is not possible to reconstruct architectures from source code fully automatically [Müller et al, 1993], several publications indicate that the architectural views reconstructed this way can serve as good starting points for manual refinement. Based on this we formulate the following hypothesis:

> *H3: Automatic clustering-based architecture reconstruction methods can reconstruct an architectural view of the Océ Controller from its source code that is a good starting-point for manual refinement.*

We use the MoJoQuality metric to quantify "good". It will be used to compare the clustering result to the result of a manual reconstruction of the architecture. The latter will be referred to as the *expert decomposition*. Paragraph 7.2.2 describes how this decomposition is obtained. The motivation for choosing the MoJoQuality metric is described in the "Assessment of output" section in paragraph 7.2.1. Based on clustering results reported in literature[26], we consider a decomposition produced by architectural clustering good if it has a MoJoQuality of at least 60% relative to an expert decomposition.

During its lifetime the Océ Controller has been modified extensively. As described in paragraph 2.1.3, the internal structure of software systems that are continuously modified inevitably deteriorates, which obfuscates the architecture. This means that in the original version the architecture is present in a purer form than in later versions. Since architecture reconstruction is usually performed for software of which several versions have been released, it is likely to be applied to software of which the architecture has deteriorated significantly. We speculate that this reduces the effectiveness of clustering-based architecture reconstruction techniques. If this is the case, incorporating information from multiple versions in the clustering process could improve the quality of the result. This leads to the following hypothesis:

> *H4: Utilizing information obtained from source code of older versions can improve the quality of the output of architectural clustering algorithms for more recent versions of a system.*

The "Combining version information" section in paragraph 7.2.1 describes how the information of the older versions is used in the clustering process.

An architectural clustering is considered to be better than another clustering if it achieves a lower EdgeMoJo value. The EdgeMoJo metric is used here because it gives a more detailed assessment of a decomposition's quality than MoJoQuality. Because it produces a non-normalised result it cannot be used for H3 however. The "Assessment of output" section in paragraph 7.2.1 describes the reasons for choosing this metric in more detail.

[26] More precisely, we base this on [Tzerpos and Holt, 1999], [Wen and Tzerpos, 2004b] and [Trifu, 2003].

A workbench has been built that uses clustering techniques to reconstruct a static view of a software architecture from source code. This workbench can incorporate information obtained from the source code of multiple versions of a system into the clustering process. The workbench is applied to the Océ Controller to confirm the two hypotheses.

7.2 Architectural-clustering architecture

This paragraph discusses the architecture of the architectural-clustering workbench. Before the actual architecture is described, the decisions that led to it are discussed.

7.2.1 Initial Choices

The following issues need to be considered in a clustering task [Jain et al, 1999]:
1. Entity representation.
2. Similarity metric.
3. Algorithm choice (how to group the entities).
4. Data abstraction.
5. Assessment of output.

This paragraph describes how the architectural-clustering workbench handles these issues. Paragraph 6.1 gives a detailed description of the nature of these issues.

Entity representation & feature selection

Most researchers agree that architectural clustering approaches based on structural criteria and naming conventions are the most promising ones [Tzerpos and Holt, 2000]. Recall from paragraph 1.2 that most of the Océ Controller is written in an object-oriented programming language (C++). In object-oriented software classes are the most important building blocks [Booch et al, 1999]. They provide an initial grouping of closely related data and functions [Mitchell, 2002]. Architectural clustering approaches for object-oriented source code reported in literature often choose classes as the entities to be clustered. We therefore decided that the set of classes extracted from the source code would form the entity set. Clusters grouping a number of classes will be called *subsystems*, or simply clusters.

Based on [Tzerpos and Holt, 2000], [Mitchell, 2002] and [Trifu, 2003] we decided that the clustering will be based on structural relations between the classes. Other approaches use different kinds of information, like for example ownership information, as is described in paragraph 3.5.4. However, this kind of information is not always available. If the complete[27] source code of a system is available, all structural relations between classes are available. Because this is the minimal amount of information that must be available for architecture reconstruction to make sense, an approach based on this kind of information has the widest applicability.

We distinguish the three most important types of relationships between classes in object-oriented systems [Booch el at, 1999]:
- **Association**: a structural relationship between two classes that specifies one class is connected to another.
- **Generalization**: the object-oriented mechanism via which more specific classes incorporate the structure and behaviour of more general classes.
- **Dependency**: a “using” relationship that specifies a change in one class may affect another class.

If two classes are related by any of these relations, it is possible that multiple instances of this relation exist. For example if a class c_1 has three methods that all reference a class c_2, these lead to three dependencies from c_1 to c_2. The clustering can take the actual number of relation instances into account, or just its presence. To our knowledge no work has been published describing the effect of this choice on the quality of the clustering result in the context of object-oriented software. We therefore decided to define a user-specified

[27] Here “complete” means that a working system can be compiled from the source code.

parameter p_c (c for combine), that specifies if only the presence, or also the number of instances of a relation between two classes must be taken into account:

- $p_c \equiv false \Rightarrow$ take the number of instances of each class-relation into account.
- $p_c \equiv true \Rightarrow$ take only the relation's presence into account, not the number of instances.

In various publications it is suggested to use different weights for the different relationship-types, making certain types more important than others [Mitchell, 2002], [Trifu, 2003].
However, to our knowledge no work exists that describes how different choices for the weights of the object-oriented structural relation-types affect the quality of the clustering result. We therefore decided to introduce a user-specified parameter for each relationship-type that specifies the weight of instances of this relation in the similarity calculation. This leads to three parameters, p_{w_a}, p_{w_g} and p_{w_d}, which specify the weight of association, generalization and dependency relations respectively.

Similarity metric and algorithm choice

Given the entity representation and feature selection chosen in the previous paragraph, a clustering algorithm is required that can cluster an entity set with inter-entity features[28]. Bunch [Bunch, 2005] is a tool that implements several clustering algorithms that operate on this kind of data. It has been used in various architectural-clustering experiments and is known to produce clusterings within a bounded approximation of the optimal clustering [Shokoufandeh et al, 2004]. Because Bunch has been implemented as a generic clustering tool, it can easily be integrated in an architecture-reconstruction workbench.

Because Bunch seemed the most appropriate choice, we decided to use Bunch in the architectural-clustering workbench. Based on experiences with Bunch reported by [Mitchell, 2002], we decided to use the hill-climbing algorithm and the TurboMQ similarity metric. Paragraph 6.2.4 describes this algorithm, the TurboMQ metric, and several applications of Bunch.

Note that we use Bunch differently than the applications reported in literature. The difference is threefold:

- We use Bunch to cluster object-oriented software, whereas the applications reported in literature cluster procedural code[29]. This affects the entity representation and feature selection, and not the clustering algorithm itself. Hence, this does not invalidate the conclusions of [Shokoufandeh et al, 2004].
- We distinguish multiple different relationship-types with different weights, whereas the applications reported in literature use the same weights for all relationship-types. However, Bunch has been designed to support different weights for the relationships. A small experiment revealed that the weights have the expected effect on the clustering result.
- Applications reported in literature use information from one version. In some cases we use information from multiple versions, as is described in the "Combining version information" section in this paragraph. This however only affects the number of features, and not the clustering algorithm itself. Hence, this does not affect the conclusions of [Shokoufandeh et al, 2004].

Data abstraction

Bunch automatically generates names for the created clusters. These names are based on an increasing sequence number and the level of the cluster in the decomposition. However, these names have little meaning to software maintainers. Ideally the workbench gives meaningful names to the clusters. Because we consider the decomposition produced by the architectural clustering as a starting point that needs to be refined manually, using the names Bunch generated is no significant restriction. Therefore we decided to leave the issue of

[28] These are features that describe relationships between the entities, as is described in paragraph 6.1.3.
[29] As described in paragraph 6.2.4 those approaches use source files as entities, and dependencies between source files as features.

automatically giving meaningful names to the clusters as future work and use the names Bunch generated.

Assessment of output

As described in paragraph 6.3, architectural clustering methods usually assess their output by comparing it to some expert decomposition, or by manually checking if it *seems* appropriate. In literature consensus is that the first method is to be preferred [Mitchell, 2002], [Koschke, 2000], so we choose to implement this method in the workbench.

To validate hypothesis H3 a metric is needed that produces a normalised result that can be compared to values for similar cases reported in literature. Paragraph 6.3 describes several metrics for the comparison of automatically generated decompositions to expert decompositions. We choose to use the MoJoQuality metric [Tzerpos and Holt, 1999] because it is a normalised metric for comparing clustering results to expert decompositions for which reference values have been published.

The EdgeMoJo metric [Wen and Tzerpos, 2004a] is a non-normalised metric that uses a similar approach as MoJoQuality. Unlike MoJoQuality, this metric also takes the relations between the classes into account. Incorrect class-placements that affect many relations are considered more important than those that affect a few relations. Several recent publications concerning comparison metrics for clusterings[30] state that that it is important to take edge-information into account also. Because we agree with this, we decided to use the EdgeMoJo metric to validate hypothesis H4. Because it is not normalised, this metric cannot be used to compare the quality of architectural clusterings of different systems (so for hypothesis H3), but it can be used to determine if changes to the clustering process lead to an improvement of the result (assuming that the same classes are clustered).

Due to the size of the Océ Controller (the last version contains 2661 classes) both the expert decomposition and the decomposition Bunch produces must be hierarchical. To our knowledge the approach [Shtern and Tzerpos, 2004] described is the only method for comparing hierarchical decompositions that is reported in literature[31]. We therefore decided to use this approach to assess our decompositions.

For more information on the two similarity metrics and the conversion approach the reader is referred to paragraph 6.3.2.

Combining version information

Approach

Hypothesis H4 states that the use of information from older versions of the Océ Controller during the clustering process can improve the quality of a decomposition of the last version. In this context “improve” means that this decomposition comes closer to an expert’s decomposition.

The underlying assumption is that the original architecture of the system has deteriorated and that this reduces the effectiveness of architectural clustering.
When a system is refactored its internal structure improves again. This makes it unlikely that the “structuredness” of systems is decreasing monotonically. It is even possible that after a refactoring a system’s envisioned architecture is implemented more accurately than in the first version. Therefore it is not necessarily the first version of the system in which the architecture is present in its purest form. This must be taken into account when choosing the versions to combine.

Figure 31 illustrates the difference between architectural clustering based on a single version and on multiple versions. Let $\{V_1, \ldots, V_n\}$ be the versions of the analysed system, where V_j is

[30] For example [Mitchell and Mancoridis, 2001] and [Wen and Tzerpos, 2004a].
[31] Actually, this method describes transformations that allow using a metric for partitionings to compare hierarchical decompositions.

released before version V_{j+1} $(1 \leq j < n)$. From now on we assume that an architectural view of the most recent version, V_n, is reconstructed, since maintenance will usually be done on this version. Of each version V_i that is involved in the clustering process $(1 \leq i \leq n)$ a model M_i is constructed. Each of these models describes a single version of the system in terms of classes and the relations among them, as is described in paragraph 6.1.3.
If architectural clustering only uses information from version V_n (the last one), it uses clustering to construct an architectural model A_n from M_n. This is illustrated on the left side of Figure 31.
If architectural clustering uses information from for example versions V_1 and V_n, the models M_1 and M_n are combined, producing a model $M_{1,n}$. The construction of the architectural model A_n is then based on $M_{1,n}$. This is illustrated on the right side of Figure 31. This paragraph describes two methods for combining information from multiple versions.

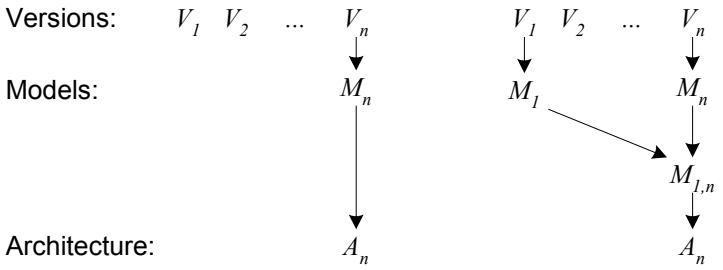


Figure 31: Architectural clustering based on a single (left) and multiple versions (right)

Before the architectural-clustering workbench can be described further, it must be described how two models of system-versions are combined. Recall that these models consist of classes and structural relations among these classes. In the remainder of this paragraph these two are addressed separately, starting with the classes.

Combining the classes

The first question is which classes are selected from the two models. Let us assume that one of the models represents the version of which the architecture is reconstructed, V_n. A decomposition that contains the *united* sets of classes of V_n and some version is likely to contain classes that are no longer present in V_n. Because this shows unexpected classes to maintainers we argue that this must be avoided.

On the other hand, a decomposition of V_n that only contains the classes that were *also* present in the other version will not have much value either, because it is likely to leave some of the classes of V_n unclassified.

We therefore decided that the produced decomposition must contain the classes in V_n and no more.

Combining the relations

The second question is how the structural information of the two models is combined. From the preceding discussion it is obvious that only the classes present in the last version must be clustered. This means that information from other versions must be incorporated through the relationships. Set operations like union and intersection can be used to combine sets, but their use leads to unexpected results, as is demonstrated below.

Let C be the set of all classes of all versions, and T the set of relationship-types between the classes. As described earlier in this chapter, multiple instances of a relationship may be present between two classes. Therefore each relation has a source, target, type and *count*. The count value represents the number of instances of the relation. The set of class-relations

R is defined such that it contains each distinct triple of a source, target and type at most once. Hence, if $\mathbb{N}$ the set of natural numbers $R \subseteq C \times C \times T \times \mathbb{N}$.

We will refer to the third component of R as the *type-component*, and to the fourth component as the *count-component*. An element of R is called a *class-relation*.

Sets of class-relations can be combined with set operations like union and intersection. Due to the count-component in R, applying these operations directly to two of these sets leads to strange results. For example, suppose that a system consists of two classes, x and y, and that only one relationship-type, say t, is present in the model. Suppose further that two versions of the system exist, V_1 and V_2, with sets of class-relations R_1 and R_2 respectively. Suppose also that in version V_1 n_1 instances of relation t exist from x to y, and in version V_2 n_2 instances $(n_1 \neq n_2)$. This gives the class-relation sets $R_1=\{(x,y,t,n_1)\}$ and $R_2=\{(x,y,t,n_2)\}$. Now $R_1 \cap R_2=\{(x,y,t,n_1)\} \cap \{(x,y,t,n_2)\}=\emptyset$. However, it is highly unlikely that this represents the set of class-relations present in both versions because the two versions probably do have some relations in common. We therefore cannot use the normal set operations to combine sets of class-relations.

Below two operations to combine sets of class-relations are described that do not have this problem. They are described using the definitions of C, T and R given earlier in this chapter. The first operation is class-relations-intersection, which intersects two sets of class-relations. The second is class-relations-union, which unites two sets of class-relations.

Class-relations-intersection, denoted by $\cap_r$, is an operation with type $R \times R \rightarrow R$ that gives the class-relations present in *both* sets of class-relations, ignoring differences in the count component. Informally, the class-relations-intersection of two sets of class-relations R_i and R_j starts by intersecting R_i and R_j with the count-component removed. Next, each tuple of the result is extended with a count-component that is the minimum of the count-components of the corresponding tuples in R_i and R_j.
If $n_i \downarrow n_j$ refers to the minimum of two values n_i and n_j, the class-relations-intersection of two sets of class-relations $R_i, R_j \subseteq R$ is defined as:

$$R_i \cap_r R_j = \left\{ \left(x,y,t,n_i \downarrow n_j\right) \middle| \left(x,y,t,n_i\right) \in R_i \wedge \left(x,y,t,n_j\right) \in R_j \right\} \tag{35}$$

Consider the following example. Suppose that the set of classes $C=\{x_1,x_2,x_3\}$ and the set of relationship-types $T=\{a\}$. Further suppose we have two sets of class-relations, namely $R_3=\{(x_1,x_2,a,3),(x_1,x_3,a,1)\}$ and $R_4=\{(x_1,x_2,a,1),(x_2,x_3,a,3)\}$. Then
$R_3 \cap_r R_4=\{(x_1,x_2,a,1)\}$.

Class-relations-union, denoted by $\cup_r$, is an operation with type $R \times R \rightarrow R$ that gives the class-relations present in *any* of the two sets. In general, when uniting sets, two types of elements can be distinguished: those present in both sets, and those present in one but not in both sets. The class-relations-union operator treats these two types differently. First, the class-relations union of two sets of class-relations R_i and R_j calculates the (normal) intersection of the two sets without the count-component, and adds a count-component to each tuple that is the *maximum* of the count-components of the corresponding tuples in R_i and R_j. Second, the obtained set is extended with the tuples in R_i for which no corresponding tuple in R_j exists and vice versa.

In order to define the class-relations-union operator more precisely, we need an operator to test the membership of an element in a subset of R without considering the count-component. We therefore define the class-relations-membership operator $\in_r$. This operator takes an element (x,y,t,n) of R and a subset $R_x \subseteq R$, and gives either true or false. If $\mathbb{N}$ is the set of natural numbers, it is defined as:

$$\left(x,y,t,n\right) \in_r R_x \Leftrightarrow \left(\exists m \in \mathbb{N} : \left(x,y,t,m\right) \in R_x\right) \tag{36}$$

The class-relations-membership operator gives true if the provided set contains an element that equals the provided element on the first three components. Otherwise this operator gives

false. For example suppose that we have the sets of classes and relationship-types defined above and $R_5=\{(x_1,x_2,a,3)\}$. Then $(x_1,x_2,a,1)\in_r R_5$ is true and $(x_1,x_3,a,3)\in_r R_5$ is false.

If $n_i\uparrow n_j$ refers to the maximum of two values n_i and n_j, the class-relations-union of two sets of class-relations $R_i,R_j\subseteq R$ is defined as:

$$R_i \cup_r R_j = \left\{\left(x,y,t,n_i \uparrow n_j\right)\middle|\left(x,y,t,n_i\right)\in R_i \wedge \left(x,y,t,n_j\right)\in R_j\right\} \cup\left\{r\in R_i\middle|r\in_r R_j\right\}\cup\left\{r\in R_j\middle|r\in_r R_i\right\} \quad (37)$$

For example, suppose that we have the sets of classes and relationship-types defined earlier and $R_6=\{(x_1,x_2,a,3)\}$ and $R_7=\{(x_1,x_2,a,1),(x_2,x_3,a,3)\}$. Then
$R_6\cup_r R_7=\{(x_1,x_2,a,3),(x_2,x_3,a,3)\}$.

7.2.2 Workbench architecture

Based on the decisions described in the previous paragraph the architecture of the architectural-clustering workbench can now be defined. Figure 32 shows a conceptual view of it. The boxes indicate processing steps and the black arrows directed dataflows.

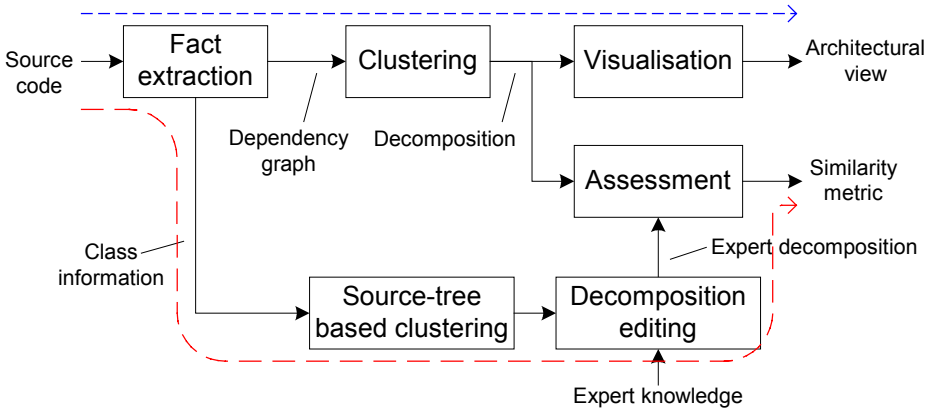


Figure 32: Conceptual view of the workbench

Figure 32 illustrates two typical usage scenarios of the workbench:
1. Automatic generation of a decomposition with clustering (fine dotted blue arrow).
2. Assessment of the clustering result (coarse dotted red arrow).

Both scenarios start with the extraction of facts from the source code. The first scenario represents the normal process when using clustering to reconstruct an architecture from source code. In this scenario the dependency graph that was extracted from the source code is clustered and the result is visualised.

The second scenario is a validation scenario that is used to validate the approach. In this scenario the clustering result is compared to an expert decomposition. This decomposition is obtained in two steps. First the classes found during the fact extraction step are organised according to their location in the source tree. Although our reconstruction approach does not need this information, in the case of the Océ Controller it is available and not using it would make the manual construction of the expert decomposition much more labour intensive. Second, an editor is used to refine the resulting “draft” decomposition. The resulting expert decomposition is then compared to the clustering result.

Figure 33 shows a process view of the workbench architecture. The rectangles represent processes and the black arrows directed communication channels. The dotted lines represent the data flows of the two scenarios discussed above.

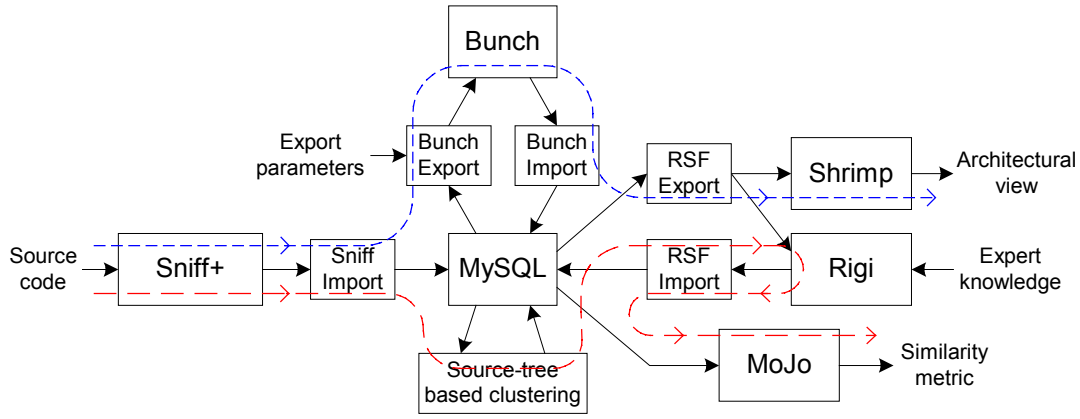


Figure 33: Process view of the workbench

The Sniff+ module extracts the facts from the source code. Because of the large size of the Océ Controller's and the experiences in the previous case study (see chapter 5), it is expected that this is a computation-intensive step. Therefore we decided to store the results in a database, in this case MySQL [MySQL, 2005a]. The Bunch module implements the clustering process. During the conversion of the facts into a format Bunch accepts the Bunch Export module takes several user-specified parameters into account. This is discussed in more detail in the "Bunch Export, Bunch & Bunch Import" section of this paragraph.

The expert decomposition is obtained in the two steps shown in Figure 32. First, the classes are organised according to the structure of the source tree. This is implemented in the module labelled "source-tree based clustering". We expect this to be a reasonable approximation of the expert decomposition. In the second step an expert uses Rigi to refine this approximation.

The Shrimp module allows users to browse a decomposition, but without editing possibilities. The last module, labelled MoJo, implements the comparison of two decompositions to assess the quality of the clustering result.

The import and export modules contain "glue-logic" that connects the third party applications to the database. Because of the limitations of the XSLT language encountered in the first case study, a different language is required. Java [Java, 2005] is chosen because it is a mature and stable programming language, and because the most important third party applications used in the workbench (MySQL, Sniff+, Bunch and MoJo) have a Java API.

The next paragraphs describe each of these modules in more detail. Before the actual modules are discussed, the meta-model used for MySQL is discussed. This model is discussed separately because it affects all modules.

Meta-model

The model

The meta-model of the workbench is an abstraction of the source code from which the input for the clustering process is derived. It needs to accommodate the classes and the three types of structural relations among them that were identified in the "Entity representation & feature selection" section in paragraph 7.2.1. Since the clustering uses information from multiple versions of the Océ Controller, the model must accommodate multiple versions.

Based on existing meta-models of architecture reconstruction workbenches[32], the meta-model for the workbench has been defined. Figure 34 shows an ER model [Silberschatz et al, 2002] of the result. In this figure the rectangles represent (ER) entity-sets, diamonds relationships

[32] More precisely, this model is based on the FAMIX [Bär et al, 1999], MeMoJ [Bauer and Trifu, 2004], Columbus [Columbus, 2003] and HisMo [Ducasse et al 2004] meta-models.

between entity-sets and ellipses attributes. For notational convenience lines are used to represent one-to-one and one-to-many relationships (the numbers denote the cardinality).

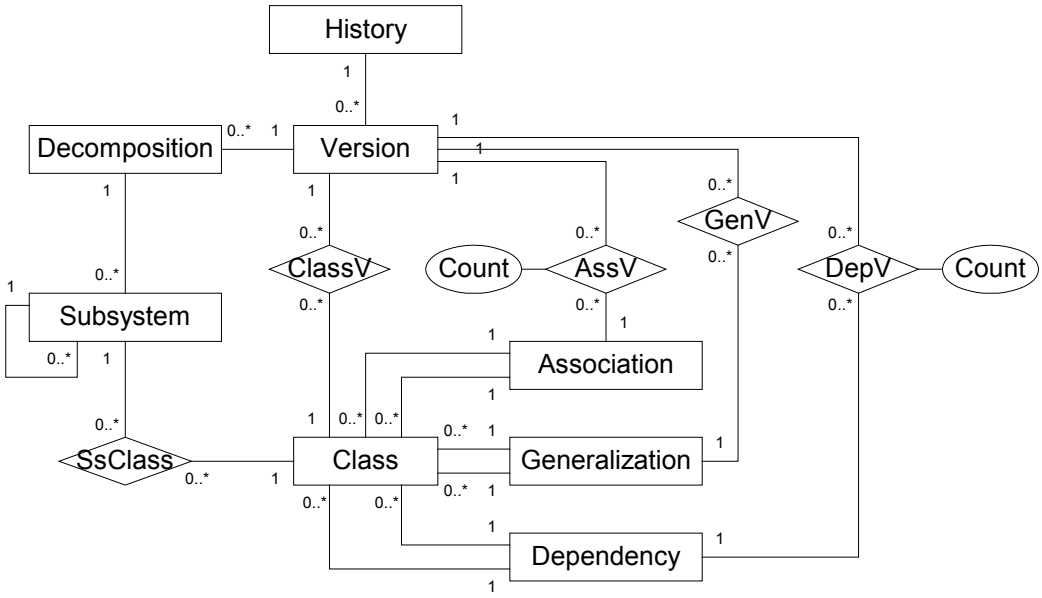


Figure 34: Clustering workbench meta-model

The entities that are clustered (the classes) are the central component of the meta-model. In the model a class is associated with certain versions of the system, indicating its presence in these versions. A set of versions forms a history of a system.

As described in paragraph 7.2.1 three types of relationships between classes are distinguished:

- Associations
- Generalizations
- Dependencies

In the meta-model these are represented by the corresponding entities. Each relationship is associated with two classes, and is present in one or more versions. Because multiple association and dependency relations can exist between two classes, a numeric “count” value is associated with these relations. An alternative would be to allow duplicate tuples in the association and dependency relations, but that would increase the space complexity of instances of the model significantly without adding any information.
The count-value is not relevant for the generalization relationship because by definition only one such relationship can exist between two classes.

The right side of Figure 34 models decompositions of the analysed system. A decomposition classifies classes of a certain version of the system. It contains a set of subsystem-trees, which is modelled by the recursive relation of the subsystem entity. Each subsystem groups a set of classes.

Relation schemas

Based on the ER model in Figure 34 the schemes of the clustering workbench’s tables are defined. The left column in Table 19 shows them using the notation [Silberschatz et al, 2002] introduced. In this notation the underlining indicates primary keys. The right column in Table 19 shows abbreviated names of the relations, which are used in the remainder of this chapter. For example the Class relation will be referred to as the C relation and the ClassV relation as CV.

Association	(ID, SourceID, DestinationID)	$A\ (id,sid,did)$
AssV	(AssID, VersionID, Count)	$AV\ (aid,vid,\ c)$
Class	(ID, Name, Scope, SourceFile)	$C\ (id,n,s,f)$
ClassV	(ClassID, VersionID)	$CV\ (cid,vid)$
Decomposition	(ID, Description, BaseVersionID)	$DC\ (id,d,bid)$
Dependency	(ID, FromID, ToID)	$D\ (id,fid,tid)$
DepV	(DependencyID, VersionID, Count)	$DV\ (did,vid,c)$
Generalization	(ID, ParentID, ChildID)	$G\ (id,pid,cid)$
GenV	(GeneralizationID, VersionID)	$GV\ (gid,vid)$
History	(ID, Description)	$H\ (id,d)$
SsClass	(ClassID, SubsystemID)	$SC\ (cid,sid)$
Subsystem	(ID, DecompositionID, ParentID, Name)	$S\ (id,did,pid,n)$
Version	(ID, Description, HistoryIndex, HistoryID)	$V\ (id,d,hix,hid)$

Table 19: Clustering workbench relation schemes

Notation

Relational algebra [Silberschatz et al, 2002] will be used to refer to the meta-model. This algebra uses the σ, Π, $\cup$ and $\bowtie$ operators to refer to selection, projection, union and inner join respectively. To illustrate this we give a few examples:

- $\sigma_{vid=5}(CV)$ selects the tuples from the CV relation that have the vid attribute set to 5.
- $\Pi_{id}\ (C \bowtie_{C.id=CV.cid} \sigma_{vid=5}(CV))$ takes the inner join of the C relation and a subset of the CV relation (the tuples with $vid=5$), and projects the id attribute of the resulting relation. This expression gives the set of IDs of classes present in version 5. The $C.id=CV.id$ predicate is called the *join condition*. If the join condition is omitted a join is performed on the attributes in the two relations that have the same name.
- ${}_{aid}\mathcal{G}_{sum(c)\ as\ d}\ (AV)$ is an example of an aggregation operation. It gives a relation with two attributes, aid and d, such that d contains the summation of the AV relation's c attribute when this relation is grouped by the aid attribute.

A small subset of tuple relational calculus will also be used. In this notation $t[a]$ denotes the value of attribute a of tuple t and $r \in R$ denotes a tuple r in relation R. For example:

- $t \in C \wedge t[id]=7$ refers to a tuple t in the C relation of which the id attribute is 7.
- $tv \in CV \wedge tv[cid]=4 \wedge tv[vid]=2$ refers to a tuple tv in the CV relation of which the cid attribute is set to 4 and the vid attribute to 2. This tuple indicates that the class with ID 4 is part of the version with ID 2.

For a more complete description of tuple relational calculus and relational algebra the interested reader is referred to [Silberschatz et al, 2002].

MySQL

As described in the previous paragraph the database in the workbench stores the facts that are extracted from the source code and the created decompositions. MySQL is a relational database that claims to be *"the world's most popular open-source database with over six million installations"* [MySQL, 2005a].
The workbench uses the MySQL database because it is a stable, proven database. MySQL Connector /J [MySQL, 2005b] is used to interface with the import and export modules.

During the installation MySQL is configured for optimal performance on large databases with a small number of concurrent clients. To eliminate the performance penalty induced by transaction support (logging mechanisms, see [Silberschatz et al, 2002]), the import and export modules do not use transactions. This is possible because only one module is active simultaneously.

MySQL supports several different storage engines. The workbench uses the MyISAM storage engine for all tables in the meta-model. This engine is chosen because it is the fastest engine that provides persistent storage [Lentz, 2004]. Temporary tables are stored with the MEMORY storage engine, which is even faster than the MyISAM storage engine, but does not provide persistent storage when MySQL is shut down.

Sniff+ & Sniff Import

In the first case study, Columbus/CAN is used for fact extraction. As is described in the "Encountered problems" section in paragraph 5.4.3, Columbus/CAN is not able to extract facts from the complete Océ Controller. Therefore a different fact extractor is required for this case study. The Sniff+ parser is used in several reverse engineering frameworks and performs well in various comparisons of reverse engineering tools (see paragraph 3.3.6). Further, Sniff+ has a Java API that provides access to its parsing results. Therefore it is decided to use Sniff+ for fact extraction.

In the workbench the Sniff Import module uses the Sniff API to extract facts from the source code and write these to the database. This API is based on a publish-subscribe model. Subscribers register classes that contain handler functions for various entity-types. Sniff+ contains a publisher module that calls these methods during the extraction. Examples of such classes are the SymbolHandler and the ReferenceHandler classes.

Sniff+ identifies entities like classes and methods with an internal identifier. However, this identifier is not unique across multiple Sniff+ sessions. Because the different versions are extracted from the source code in separate Sniff+ sessions, a new unique identifier for the classes is required. The Sniff Import module bases this identifier on the id attribute of the C relation, which is an automatically generated primary key.

If multiple versions of a system are analysed and one version has already been loaded in the database, subsequent versions can contain the same classes. During the extraction of these versions only new classes are added to the C relation. This however requires a unique identification of the classes. Class-names and scopes do not provide a unique identification for the classes for two reasons:

- The Océ Controller consists of multiple executables. Because these executables are compiled separately this allows duplicate class names and scopes to be used.
- Some branches of the source tree are nearly identical copies of other branches. Examples are stubs, nearly identical test tools, and experimental copies of subsystems that are stored for future use. In these cases, based on configuration- and make-files, the build-process selects certain branches of the source-tree. Sniff+ does not take the configuration- and make-files into account, producing duplicate class-names and scopes.

Within a single version of the Océ Controller a combination of the class-name and the path of its source-file can uniquely identify classes. Therefore it is decided to use the combination of the f and n attributes of the C relation to uniquely identify classes during the addition of classes to the database.

Recall that three types of relationships between the classes are extracted, namely generalizations, associations and dependencies. Let x_a and x_b be two classes with database IDs i_a and i_b respectively. Then source code constructs are mapped to these relationship types as follows:

- **Generalization**: suppose x_a inherits from x_b. This inheritance leads to a tuple $t_g \in G$ with $t_g[cid]=i_a \wedge t_g[pid]=i_b$.
- **Association**: suppose x_a has a member variable of type x_b, or a pointer to type x_b. This leads to a tuple $t_a \in A$ with $t_a[sid]=i_a \wedge t_a[did]=i_b \wedge t_a[c]=n_a$, where n_a is the number of association-instances from x_a to x_b.
- **Dependency**: suppose x_a contains a method that refers to a variable of type x_b, a method of x_b, or an attribute of x_b, such that $x_a \neq x_b$. This leads to a tuple $t_d \in D$ with $t_d[fid]=i_a \wedge t_d[tid]=i_b \wedge t_d[c]=n_d$, where n_d is the number of dependency-instances from x_a to x_b.

The GV, AV and DV relations are updated accordingly.

Dependencies from a class to itself are called *self-dependencies*. Observe that such dependencies are not extracted from the source code. This is done because self-dependencies are trivial; any well-designed class exhibits high cohesion and hence self-dependencies [Sommerville, 2004]. Furthermore, self-dependencies are not relevant for the clustering because they do not relate two different entities.

Associations of a class to itself are called *self-associations*. Unlike self-dependencies they are extracted from the source code because of their relevance at the architectural level; they indicate recursive object structures. Like self-dependencies, self-associations are not relevant for the clustering process.

Bunch Export, Bunch & Bunch Import

The Bunch Export module creates a file containing the module dependency graph (MDG). Bunch reads the entities to be clustered and the relations between them from this file. After the clustering process is completed, the Bunch Import module writes the produced decomposition to the database. Note that the MDG-file is not limited to the dependency relationship type, but can contain any kind of relationship.

Bunch Export

Each line in the MDG-file written by the Bunch Export module specifies a single edge in the module dependency graph. Together, the lines in the MDG-file specify the bag of edges E of the module dependency graph $G=(N,E)$, where N represents the set of nodes (the classes). N is not specified explicitly in the MDG-file, but implicitly by E.
Let $x_1,x_2 \in N$ be two classes. A line in the MDG-file in the following format represents a relation from x_1 to x_2 with weight w [Mitchell, 2002]:

$$x_1\ x_2\ w$$

The workbench consecutively writes the contributions of the associations, generalizations and dependencies to the MDG-file to obtain E. A mapping between the meta-model of the workbench and these three contributions, E_a, E_g and E_d respectively is defined below.

Bunch Export: queries for one version

Let v be the ID of the version that must be clustered. The $E_a(v)$ relation contains the contribution of the associations to E for version v. It is obtained by joining relation A with the subset of the AV relation that is related to version v. More precisely, $E_a(v)$ is calculated by joining the tuples in A with $sid \neq did$ to the tuples in AV with $vid=v$. Tuples in A with $sid=did$ (self-associations) are removed because Bunch cannot handle edges that start and end in the same node. Because such relations have no effect on the clustering, this does not affect the clustering result. Finally, a projection selects the appropriate attributes:

$$E_a(v) \leftarrow \Pi_{sid,did,w_a}\left(\sigma_{sid \neq did}(A) \bowtie_{id=aid} \sigma_{vid=v}(AV)\right)$$
$$\text{with } \begin{cases} w_a = c \times p_{w_a} & \text{if } \neg p_c \\ w_a = p_{w_a} & \text{if } p_c \end{cases} \quad (38)$$

In (38) p_c is the user-specified parameter described in paragraph 7.2.1 that specifies if the number of instances, or just the presence of a relationship between two classes must be considered. Further, p_{w_a}, is a user-specified parameter that defines the weight of association relations (see paragraph 7.2.1 also). Observe that the edges point from the child class to its parent. This direction is chosen because the child depends on the parent and not the other way around.

The $E_g(v)$ relation contains the contribution of the generalizations to E for the version identified by v. It is obtained in a similar way as $E_a(v)$, namely by selecting the subset of relation GV that is related to the version v identifies, and joining the resulting relation with the G relation:

$$E_g(v) \leftarrow \Pi_{cid,pid,p_{w_g}} \left(G \bowtie_{id=gid} \sigma_{vid=v}(GV) \right) \tag{39}$$

GV has no c attribute because by definition only a single generalization relation can exist between two classes. Analogue to p_{w_a}, p_{w_g} is a user-defined parameter that specifies the weight of generalization relations.

The $E_d(v)$ relation contains the contribution of the dependencies to E for the version identified by v. The calculation of this relation is slightly more complex than that of the previous relations. By our definition of the relationship types, an association from class x_1 to class x_2 also implies a dependency from x_1 to x_2. A similar argument can be held for generalizations: If x_1 inherits from x_2, x_1 is likely to use methods or attributes of x_2. Therefore this generalization usually leads to a dependency from x_1 to x_2.
[Booch et al, 1999] state that such redundant dependencies can be omitted from UML models because associations and generalizations imply a dependency. To our knowledge no work has been published on how the omission of these relations affects the output of architectural clustering. We therefore define a user-specified boolean parameter p_i (i for ignore) such that:

- $p_i \equiv false \Rightarrow E_g(v)$ includes all dependencies in the version.
- $p_i \equiv true \Rightarrow$ a dependency d is only included in $E_g(v)$ if
  - $E_a(v)$ contains no association a with $a[sid] = d[fid] \wedge a[did] = d[tid]$ and
  - $E_g(v)$ contains no generalization g with $g[cid] = d[fid] \wedge g[pid] = d[tid]$.

In the case $p_i \equiv false$, the $E_d(v)$ relation is obtained by joining the D relation with the subset of the DV relation that is related to the version identified by v, similar to the calculation of $E_a(v)$ and $E_g(v)$. However, if $p_i \equiv true$, some dependencies must be ignored. This is achieved by subtracting the sets of associations and generalizations from the set of dependencies.
If p_{w_d} is a user-defined parameter that specifies the weight of dependency relations, analogue to p_{w_a}, $E_d(v)$ is defined as:

$$E_d'(v) \leftarrow \Pi_{fid,tid,w_d} \left(D \bowtie_{id=did} \sigma_{vid=v}(DV) \right)$$
$$\text{with} \begin{cases} w_d = c \times p_{w_d} & \text{if } \neg p_c \\ w_d = p_{w_d} & \text{if } p_c \end{cases}$$

$$E_d(v) \leftarrow \begin{cases} E_d'(v) & \text{if } \neg p_i \\ E_d'(v) \bowtie \begin{pmatrix} \Pi_{fid,tid}(E_d'(v)) - \\ \Pi_{sid \text{ as } fid, did \text{ as } tid}(E_a(v)) - \\ \Pi_{cid \text{ as } fid, pid \text{ as } tid}(E_g(v)) \end{pmatrix} & \text{if } p_i \end{cases} \tag{40}$$

In (40) the $E_d'(v)$ relation is obtained by joining the D relation with the subset of the DV relation that is related to the version v identifies. For the case $p_i \equiv false$, $E_d(v)$ equals $E_d'(v)$. For the case $p_i \equiv true$, $E_d(v)$ only contains tuples in $E_d'(v)$ for which no corresponding tuple in $E_a(v)$ or $E_g(v)$ exists. This is calculated by projecting the fid and tid attributes of $E_d'(v)$, and subtracting the projections of the sid and did attributes of $E_a(v)$, as well as that of the cid and pid attributes of $E_g(v)$. The resulting set of tuples with schema (fid,tid) is joined with $E_d'(v)$ to obtain the c attribute.

The translation of the expressions in (38), (39) and (40) into SQL queries is trivial and is omitted from this thesis.

Bunch Export: queries for intersection of versions

Class-relations from multiple versions can be combined with the class-relations-intersection $\cap_r$ and class-relations-union $\cup_r$ operators defined in paragraph 7.2.1. In both cases the contributions of the associations, generalisations and dependencies are written to MDG-file consecutively. This is equivalent to the definitions in paragraph 7.2.1 because these three sets are disjoint (they have a different type attribute).

The class-relations-intersection operator $\cap_r$ gives the relations present in both versions and is implemented with a set of SQL queries. Recall that according to (35), $R_i \cap_r R_j$ consists of the tuples (x,y,t,c) for which a pair of tuples exists in the sets of class-relations R_i and R_j that are equal to each other on the first three components. The c component of the resulting tuples is set to the minimum value of the c component of the two corresponding tuples in R_i and R_j.

The implementation of the class-relations-intersection operator is based on the queries for one version. Since all three relationship-types are treated in the same way we only describe the implementation for the associations here. This implementation is based on the $E_a(v)$ relation (38) defines. Because the implementation handles each relationship-type separately this relation has the schema (sid,did,c).
Let u and v be the IDs of the two versions of which the class-relations-intersection is calculated. The implementation starts with projecting the sid and did attributes of $E_a(v)$ and $E_a(u)$ in order to remove the c attribute. Next the resulting relations are intersected to obtain the relation $E_{i,a}'(u,v)$, which contains the associations present in both versions. Then the implementation obtains the minimum value of the count component for every association. This is calculated by uniting $E_a(v)$ and $E_a(u)$, grouping the resulting tuples on $\{sid,did\}$, and selecting the tuple with the minimum value of the c attribute of each group. The result is joined with $E_{i,a}'(u,v)$ to select only those associations that are present in both $E_a(v)$ and $E_a(u)$.

Using the definition of $E_a(u)$ and $E_a(v)$ in (38), the contribution of the associations to the class-relations-intersection $E_{i,a}(u,v)$ of two versions u and v is defined as:

$$\begin{aligned} E_{i,a}'(u,v) &\leftarrow \Pi_{sid,did}\left(E_a(u)\right) \cap \Pi_{sid,did}\left(E_a(v)\right) \\ E_{i,a}(u,v) &\leftarrow \left({}_{sid,did}\mathcal{G}_{\min(c)}\left(E_a(u) \cup E_a(v)\right)\right) \bowtie E_a'(u,v) \end{aligned} \tag{41}$$

$E_{i,g}(u,v)$ and $E_{i,d}(u,v)$ are calculated likewise for the generalization and dependency relations respectively.

Bunch Export: queries for union of versions

The class-relations-union operator $\cup_r$ gives the relations present any of the two versions and is implemented with a set of SQL queries. (37) defines the $\cup_r$ operator as the union of three subsets. Let s_1, s_2 and s_3 be these subsets, numbered ascending from top to bottom and left to right:

- Subset s_1 is defined as the tuples (x,y,t,c) for which a pair of tuples exists in R_i and R_j that are equal to each other on the first three components. The c component of the tuple in s_1 is set to the *maximum* of the c components of the corresponding tuples in R_i and R_j.
- Subset s_2 is defined as the tuples $(x,y,t,\mathbf{c})$ from R_i for which no tuple $(x,y,t,\mathbf{n})$ exists in R_j (for any $\mathbf{n} \in \mathbb{N}$).
- Subset s_3 is obtained in the same way as s_2, but with R_i and R_j exchanged.

Observe that s_1, s_2 and s_3 are disjoint by definition.

Similar to the implementation of the previous operator the implementation of the class-relations-union operator handles each relationship-type separately. Because s_1, s_2 and s_3 are disjoint sets (without duplicates of course) that are united to get the result, taking the maximum can *after* the union instead of *before* it does not affect the result. Let w and v be the IDs of the versions of which the class-relations-union is calculated. The implementation unites

$E_a(w)$ and $E_a(v)$, groups the result on $\{sid,did\}$ and takes the tuple with the highest value of the c attribute of each group. This produces the $E_{u,a}(w,v)$ relation, which contains the contribution of the associations to the class-relations-union of version w and v:

$$E_{u,a}(w,v) \leftarrow {}_{sid,did}\mathcal{G}_{\max(c)}\left(E_a(w) \cup E_a(v)\right) \quad (42)$$

$E_{u,g}(w,v)$ and $E_{u,d}(w,v)$ are calculated likewise for the generalization and dependency relations respectively.

Clustering the MDG

After the Bunch Export module created the MDG-file, Bunch clusters it. For the experiments that showed the quality of the decompositions produced by Bunch, [Shokoufandeh et al, 2004] used the default settings. The architectural clustering workbench does this too, with one exception. In the workbench, Bunch is configured to generate "tree format" output. Activating this setting does not affect the clustering itself, but only the format of the output. If a tree format is chosen the complete dendrogram, instead of just one slice, is written to the output-file [Mitchell, 2002]. This gives a hierarchical decomposition, instead of a partitioning. If no tree format is chosen each output file contains a slice of the dendrogram and the output files together contain the complete dendrogram.

Bunch Import

The Bunch Import module reads the clustering result from the output-file written by Bunch. For each cluster defined in this file a tuple is created in the S relation. The classes and clusters placed within each cluster are handled as follows:

- For each *class* with ID x in a cluster with ID s a tuple t is added to the SC relation with $t[cid]=x \wedge t[sid]=s$. This places the class in the cluster.
- For each *cluster* with ID s' in a cluster with ID s the tuple t in the S relation with $t[id]=s'$ is updated such that $t[pid]:=s$. This makes the cluster with ID s' a child of the cluster with ID s.

Recall that in the MDG-file the set of vertices of the module dependency graph is represented implicitly by the set of edges. However, some classes in the system version of which the architecture is reconstructed may not be involved in any association, generalization or dependency relation at all, for example classes that only write a configuration file. Due to the implicit definition of the vertices the clustering algorithm does not classify these classes. If such unclassified classes exist, the Bunch Import module creates a cluster in the root of the decomposition that contains the unclassified classes. This is called the "unconnected classes" subsystem.

MoJo

This module implements the assessment of the clustering output. This is achieved by converting two hierarchical decompositions into two sets of partitionings. This produces a partitioning for each level of the hierarchical decomposition. Next, the MoJoQuality and EdgeMoJo metrics are used to compare each pair of partitionings of the same level. Paragraph 6.3.2 describes the conversion process and the two metrics, and paragraph 7.2.1 the reasons for choosing this approach.

The MoJo module implements both the conversion into partitionings and the two metrics, using the implementation of [Tzerpos, 2005]. The decompositions are extracted from the database with SQL queries.

The conversion of the hierarchical decompositions into partitionings handles each of the two decompositions separately. Recall that in this process each level of the decomposition is handled separately (level is defined in paragraph 6.3.2). If l is the level considered in the current iteration, this iteration assigns all classes in a cluster with a level higher than l to the parent of this cluster at level l. Each iteration produces a decomposition consisting of the classes and the subsystem they are placed in. For a more complete description the reader is referred to paragraph 6.3.2. This paragraph describes the implementation of the approach.

Before a hierarchical decomposition is converted to a partitioning, a temporary relation TS is created with the schema $(\mathbf{s}ubsystem\mathbf{id}, \mathbf{l}evel, \mathbf{id}At\mathbf{l}evel)$. The attributes of this relation will be referred to with the bold parts of their names. For each subsystem in the decomposition a tuple exists in TS. For each tuple in TS the sid attribute represents the ID of the subsystem in the S relation, and the l attribute the level of the subsystem. The idl attribute represents the id of the subsystem at level l or less that contains the entities in the subsystem sid identifies. This is explained further in the algorithm description given below.

Let d be the ID of the decomposition that is converted. The algorithm starts with an initialisation step, in which for each tuple $s \in S$ with $s[did]=d$ a tuple t is added to TS with $t[sid]=t[idl]=s[id]$ and $t[l]=$the level of subsystem s.
Next, the algorithm iterates over the levels of the decomposition in descending order (starting with the highest level). Level 0 is excluded because it is trivial; all classes are part of the application. For each level $l\ (l>0)$ two actions are performed in the order shown below:

1. A partitioning is exported to an RSF-file. In this partitioning the classes are not placed in the subsystem identified by $TS[sid]$, but in the subsystem identified by $TS[idl]$.
2. Every tuple $t \in TS$ with $t[l]=l$ is updated such that $t[idl]$ is set to the ID of the parent of the subsystem identified by the old value of $t[idl]$. More precisely, for a tuple $s \in S$ with $s[id]=t[idl]$ (s refers to the subsystem under consideration), $t[idl]:=s[pid] \wedge t[l]:=t[l]-1$.

The above algorithm converts the hierarchical decomposition into a set of partitionings. The MoJo module applies this procedure to both decompositions. Usually these are the clustering result and the expert decomposition. If the two decompositions have a different number of levels, the deepest level of the shallower decomposition is copied.

Next, the set of relations is exported to an RSF-file and the MoJo implementation of [Tzerpos, 2005] is used to calculate the MoJoQuality and EdgeMoJo values for each level. If L is the number of partitionings after the above procedure for two hierarchical decompositions A and B, and a_l and b_l are two partitionings obtained this way for level $l\ (1 \le l \le L)$, this procedure compares a_l and b_l. For each value of l the EdgeMoJo implementation starts with the calculation of $MoJo(a_l,b_l)$. Next, the additional cost of the edges and $MoJoQuality(a_l,b_l)$ are calculated. This produces two vectors $EM=(em_1,\ldots,em_L)$ and $MQ=(mq_1,\ldots,mq_L)$ that contain the EdgeMoJo and MoJoQuality values respectively for the different levels. As [Shtern and Tzerpos, 2004] suggested the workbench uses a linear weighting schema for the levels. Therefore, $MoJoQuality(A,B)$ and $EdgeMoJo(A,B)$ are obtained with:

$$MoJoQuality(A,B) = \sqrt{\sum_{l=1}^{L}\left(\frac{1}{L} \times mq_l^2\right)} \qquad EdgeMoJo(A,B) = \sqrt{\sum_{l=1}^{L}\left(\frac{1}{L} \times em_l^2\right)} \qquad (43)$$

The computational complexity of the algorithm to calculate $MoJo(a_l,b_l)$ for two partitions a_l and b_l, is $O(n \cdot log\ n + (|a_l| + |b_l|) \cdot |a_l| \cdot |b_l|)$, where n is the number of classes that are clustered and $|a_l|$ and $|b_l|$ the number of clusters in a_l and b_l respectively [Wen and Tzerpos, 2003]. [Wen and Tzerpos, 2004a] do not describe the algorithm used to calculate the additional cost of the edges and the computational complexity of this algorithm.

Shrimp & RSF Export

The workbench uses Shrimp[33] to visualise the decompositions. Shrimp is chosen because it is designed to browse through large hierarchical information spaces, such as decompositions of large systems. The RSF Export module uses SQL queries to write a decomposition to a structured RSF-file, which Shrimp then reads. This file contains the classes of a single version, and the relations stored in the database for this version.

[33] For more information about Shrimp the reader is referred to paragraph 3.5.1.

Figure 35 shows a view of the static structure of version 8a of the Océ Controller that is created with Shrimp. In this view the small squares represent classes and the arrows relationships between the classes. All 2.661 classes are placed in a single subsystem. Three types of relations are shown: associations (1.818, in red), generalizations (2.215, in blue) and dependencies (10.663, in green). This view is a typical example of the views obtained when converting the facts extracted from the source code to an architectural view without defining higher-level abstractions. Because of the large number of classes and relations shown this view gives no insight at all in the structure of the Océ Controller.

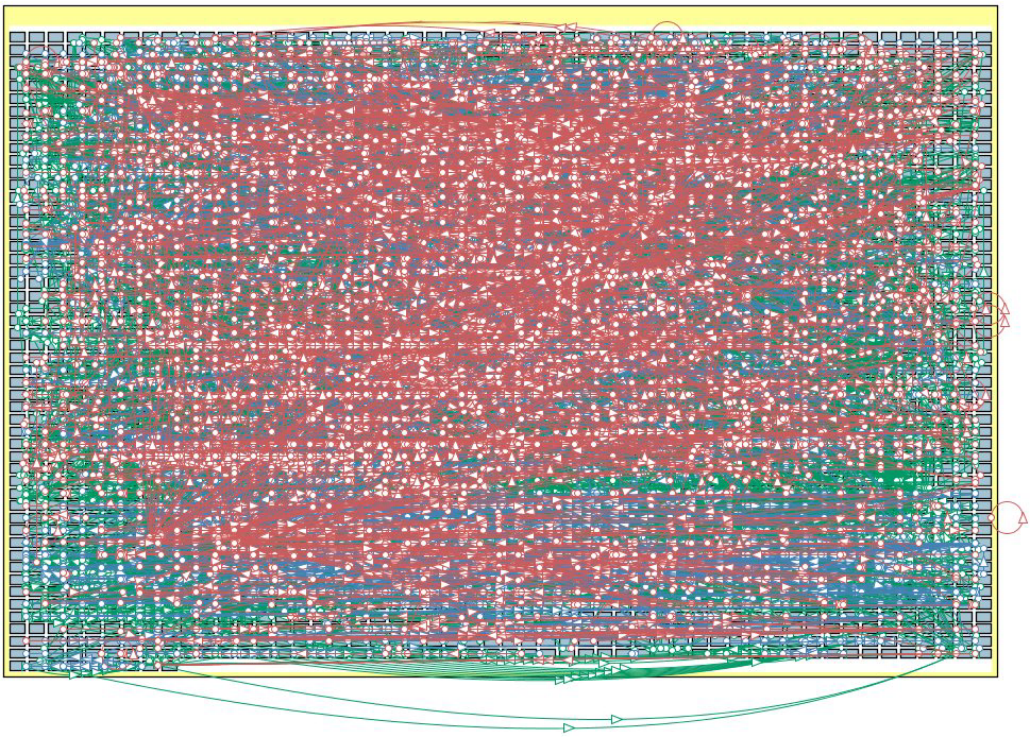


Figure 35: Initial unstructured view of the Océ Controller

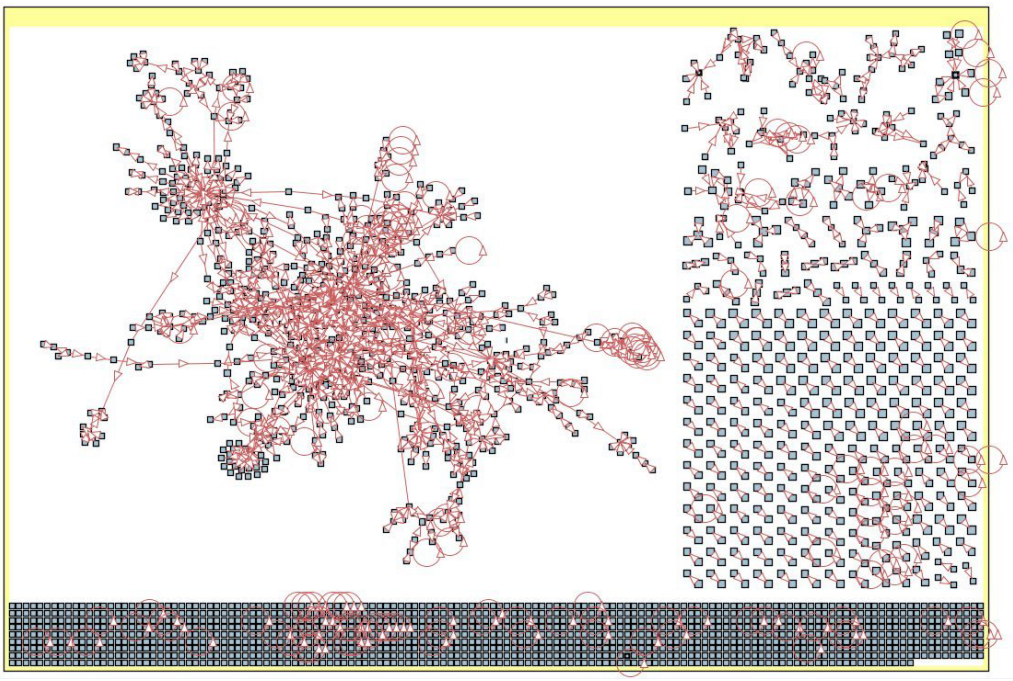


Figure 36: View of the Océ Controller based on association relations

Figure 36 shows a view of the static structure of the Océ Controller in which only the association relations are shown, in this case using a spring-layout. The arrows point from the source to the destination of the associations. Observe that several groups of related classes can be distinguished. This confirms that the association relations can be used to group the classes.

Figure 37 shows two other views of the Océ Controller that are also based on a spring-layout. On the left side a view with generalization relations is shown. The arrows point from the child to the parent class. Observe that several clusters can be distinguished here also.
On the right side a view obtained when applying a spring layout to the dependency relations is shown. The edges have been omitted here because they obfuscate the view too much. Observe that in this view the clusters are smaller than in the other views. The reason for this is that there are much more dependencies than associations or generalizations, which leads graph with a higher connectivity.

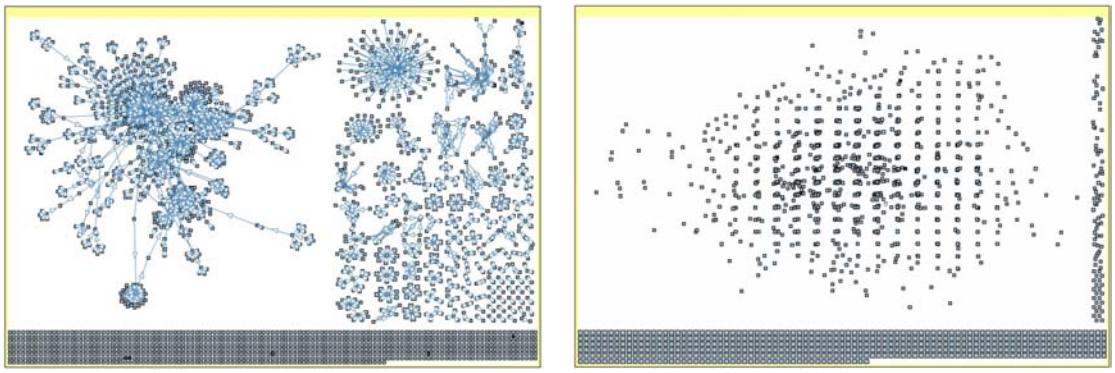


Figure 37: Views of the Océ Controller based on generalization (left) and dependency relations (right)

Source-tree based clustering

The module labelled "source-tree based clustering" (STBC) is needed for validation purposes only. It creates a decomposition for a single version based on the structure of the source tree. This decomposition is used as a starting point for the expert decomposition, which in turn is used to determine the quality of the clustering result.
The algorithm for the construction of this decomposition is based on the assumption that classes that are defined in source-files in the same directory belong together. The resulting clusters are hierarchically related by their location in the source tree. Based on approaches reported in literature[34] we expect this to be a good starting point for refinement by an expert.

The source tree of the Océ Controller contains both the source code and the documentation. If only source-files are considered this tree contains many empty directories and directories with a single subdirectory. If every directory is converted into a subsystem this is expected to lead to a poor quality decomposition because such directories do not add any structural information. Therefore they must be ignored during the construction of the decomposition. This is achieved by considering the source tree as an abstract tree, reducing it, and using the result to construct a decomposition.

The STBC module creates a directed graph $G=(N_c \cup N_d, E)$ that represents the source tree, where E represents the set of edges and $N_c \cup N_d$ the set of nodes. N_c represents the set of classes present in the version of which the architecture is reconstructed and N_d the set of distinct directories where classes in N_c are defined. Since empty directories do not define classes this definition of N_d filters out empty directories.
E contains an edge from node n_a to n_b ($n_a \in N_d,\ n_b \in N_c \cup N_d$) if and only if directory n_a contains n_b. The latter can be a directory or a class. If G has no single root node a root node

[34] More specific, we refer to [Choi and Scacchi, 1990], [Tzerpos and Holt, 2000] and [Demeyer, 2004].

n_r is created, E is extended with an edge from n_r to every node in N_d that has no parent, and n_r is added to N_d. Observe that N_d contains the interior nodes and the root node, and N_c contains the leaf nodes of the tree.

After the tree has been constructed, the algorithm reduces it by removing all nodes n_y from N_d that have a parent node n_x and a *single* child node n_z that is not a leaf node *(*so $n_z \in N_d$*)*[35]. The edges from n_x to n_y, and from n_y to n_z are removed from E, and an edge from n_x to n_z is added to it.
This is repeated until no more nodes can be removed from the tree. Then, for every interior node in the tree, a cluster is created. Each cluster created this way contains the classes and clusters that are represented by child nodes of the node representing the cluster.

After the algorithm has terminated, a subsystem is created in the S relation for every *interior* node n_i of the tree (so $n_i \in N_d \backslash \{n_r\}$, n_r refers to the root node). Let t_i be the tuple that defines a subsystem. In case n_i is not a direct child of the root node, $t_i[pid]$ is set to the value of the id attribute of the tuple in S that represents the parent node of n_i. In case n_i is a direct child of the root node, $t_i[pid]$ is set to zero. For every leaf node n_c of the tree (so $n_c \in N_c$) a tuple t_c is added to the SC relation such that $t_c[cid]$ refers to the class represented by n_c and $t_c[sid]$ to the tuple in S that represents the parent node of n_c.

On the left side, Figure 38 shows an example of G. The large circles represent nodes in N_d (directories), with their name placed inside the circle. The smaller dots represent nodes in N_c (classes). The edges represent the containment of a class or directory in another directory. The right of Figure 38 shows G after the reduction algorithm has terminated. Observe that the B, D and H nodes have been removed from the tree.

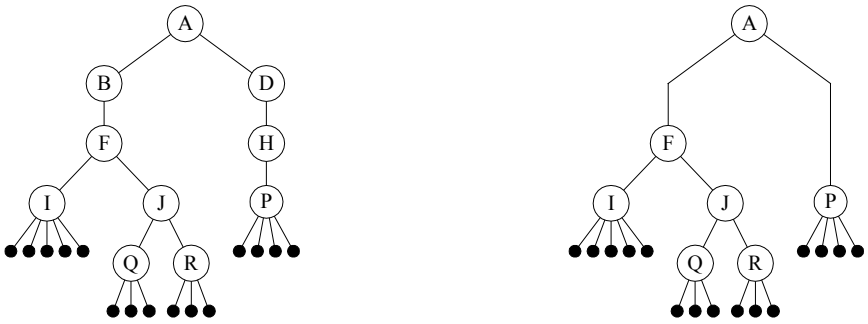


Figure 38: Example source-tree before (left) and after the reduction (right).

Rigi & RSF Import

An expert must refine the decompositions produced by the “source-tree based clustering” module. This is done with Rigi, which is described in paragraph 3.3.3.

Rigi reads the RSF-file written by the RSF Export module. Experts on the architecture then use Rigi to improve the decomposition the “source-tree based clustering” module produced, after which the RSF Import module writes the resulting RSF-file back to the database.

[35] The “no leaf node” condition is necessary for cases where only one class is defined in a directory. If a directory, say d, contains several of these directories, omitting this condition causes several classes to be placed in the cluster representing d. This is not desired because it removes too much structural information.

7.3 Implementation validation & parameter tuning

Before the workbench could be used to reconstruct the architecture of the Océ Controller it had to be ensured that its implementation was correct. Further, the proper values of the user-specified parameters had to be determined. As paragraph 7.3.2 describes we determined a set of parameter values that produced the best clusterings for two subsystems of the Océ Controller. These values were then used to cluster the complete Océ Controller. Instead of using the two subsystems we also attempted to use a specially built guinea-pig application. However, difficulties obtaining the expert decomposition to which the clustering result is compared forced us to abandon this route and use the two subsystems.

7.3.1 Validation

The validation of the implementation of the architectural-clustering workbench amounts to checking that the used third-party applications perform the expected functions and that the import and export modules apply the proper transformations. The separate modules are tested individually, or together with a few closely related modules.

Sniff and Sniff Import

The Sniff+ and Sniff Import module are tested by extracting the meta-model from the source code of several small programs with a known structure. The produced meta-model is then compared to the expected model. This procedure has been applied to two programs:

- The application used to validate the pattern detection prototype, which is described in paragraph 5.3.
- A small application based on the blackboard pattern [Buschmann et al, 1999]. This application is designed to use a wide range of source-code constructs for the association, generalization and dependency relations. This application is called the *simple-blackboard* application and is described in paragraph 7.3.2.

The tests show that all classes and class-relations are extracted from the source code, with one exception: associations created with C++ templates [Stroustrup, 1997]. Although Sniff+ can handle their use and the Sniff API has facilities for them, they are not exported. The Sniff API documentation states that template support will be added in a future version of Sniff+. However, because templates are not used very often in the Océ Controller this has little impact on the clustering result.

Bunch Export

To test the implementation of the Bunch Export module an MDG-file for the simple-blackboard application is exported. This application is described in the corresponding section in paragraph 7.3.2. Recall that five different user-specified parameters are defined that all influence the exported information:

- p_{w_a}, p_{w_g} and p_{w_d}: numeric parameters that specify the weight of association, generalization and dependency relations respectively.
- p_c and p_i: boolean parameters that reduce the amount of information that is written to the MDG-file. The p_c parameter specifies whether the instance-count or just the presence of class-relations must be written to the MDG-file. The p_i parameter specifies whether or not redundant dependencies must be omitted from the MDG-file.

The Bunch Export module is tested using three distinct values for the numeric parameters, and all possible combinations of the boolean parameters. The numeric parameters do not affect the control flow of the Bunch Export module but are written to the output directly. Therefore it is not necessary to test different combinations of the numeric parameters.

The above method is applied to the Bunch Export module, using the test application of the pattern detection prototype (see paragraph 5.3) as input. This application contains all types of relations identified in the meta-model. Further, it contains enough classes to be realistic, but not too many to make manual assessment of the output impossible.

In all cases the Bunch Export module wrote the same information to the MDG-file as was calculated manually.

Bunch, Bunch Import, RSF Export & Shrimp

Recall that the workbench uses an existing implementation of Bunch [Bunch, 2005], which has been used in various case studies. Therefore the Bunch module is only tested superficially by applying it to the simple-blackboard application and the test application for the pattern detection prototype. The produced decompositions are inspected visually with Shrimp. This is done with various values of the user-specified parameters. In all cases the produced result seemed correct.

Because these tests use Shrimp, the Bunch Import, RSF Export and Shrimp modules are also tested in this process. These three modules are tested further through the application of small test-inputs that (non-exhaustively) enumerate several input combinations. The output matched the expected result. Since the Bunch Import, RSF Export and Shrimp modules are relatively simple, this is sufficient to assess their correctness with sufficient confidence.

MoJo

Recall that the implementation of the MoJo module uses an existing implementation of the similarity metrics that was obtained from [Tzerpos, 2005]. This implementation is used in several case studies, including the ones described in [Wen and Tzerpos, 2004a]. We therefore tested this implementation only superficially. For several small, manually constructed decompositions the output equalled the values found by manual calculation of the metrics.

The implementation of the transformation of hierarchical decompositions into partitionings is tested by applying it to several decompositions and checking the resulting partitionings manually. Several decompositions Bunch produced for the test application of the pattern detection prototype served as input. The partitionings matched the expected result.

Source-tree based clustering

The source-tree based clustering module is tested by applying it to Grizzly (see paragraph 1.2). Grizzly is chosen because its source-tree is sufficiently complex to allow a realistic test, yet small enough to allow manual application of the algorithm.

The output of the source-tree based clustering module matched the decomposition that was produced manually.

RSF Export, Rigi, RSF Import

The RSF Export, Rigi and RSF Import modules are also tested by applying them to an application and checking the results manually. The simple-blackboard application is used as input because its size is sufficient for a realistic test, but manual assessment of the output is also possible. In these tests all three modules behaved as expected.

7.3.2 Parameter tuning

During the transformation of the meta-model into the module dependency graph, five user-specified parameters are used, as is described in paragraph 7.2.2. Summarizing, these parameters are:

- p_{w_a}, p_{w_g} and p_{w_d}: numeric parameters that specify the weight of association, generalization and dependency relations respectively.
- p_c and p_i: boolean parameters that reduce the amount of information that is written to the MDG-file. The p_c parameter specifies if the instance-count or just the presence of class-relations must be written to the MDG-file. The p_i parameter specifies whether or not redundant dependencies must be omitted from the MDG-file.

We call the tuple $(p_{w_a},p_{w_g},p_{w_p},p_c,p_i)$ a *parameter-tuple*.

Since these parameters directly affect the input of the clustering process, they are likely to affect the quality of its output too. To our knowledge no work has been published that describes the effect of these parameters on the clustering result in the context of object-oriented software.

As described in paragraph 1.2, the Océ Controller is relatively large. Therefore clustering its module dependency graph is relatively time consuming; on the platform described in Table 27 clustering the MDG-file of the most recent version once takes about eighteen minutes (including Bunch Export and Import). The following MoJo calculation takes about five minutes. Therefore, the number of tested parameter-tuples must be limited significantly. Besides this, the numeric parameters (that have no upper bound) make it impossible to test all different parameter-tuples anyway.

We address this issue with practical approach. First, we search for the set of near-optimal parameters for two relatively small subsystems of the Océ Controller. This allows a reasonably large area of the total search space to be investigated. Next, the parameter-tuples that lead to the best clustering for these subsystems are used to cluster the complete Océ Controller. In this process the clustering result is compared to an expert decomposition with the EdgeMoJo metric. Because the clustering algorithm Bunch uses is non-deterministic, the average EdgeMoJo value of ten different clusterings is calculated for each parameter-tuple.

Instead of using subsystems of the Océ Controller to reduce the number of parameter-tuples, a specially developed guinea-pig application can also be used. Because the Océ Controller is based on the blackboard architectural style (see paragraph 2.4.3) a simple blackboard-based application was built and it was attempted to find the set of near-optimal parameter-tuples for it. Recall that the clustering result is compared to the result of a manual decomposition. We attempted to obtain this "expert" decomposition by asking ten experienced architects and designers to reconstruct an architecture for the *simple-blackboard* application. Surprisingly, this led to ten different decompositions. Because insufficient time was available to devise a method for combining these decompositions into one, it was decided to use subsystems of the Océ Controller instead. The next section describes the "simple-blackboard" application and the experiment to obtain an expert decomposition for it.

Simple-blackboard application

Ten experienced software architects and designers were asked to reconstruct an architecture for a specially-built guinea-pig application. They based this on information that is similar to the information available in practical architecture-reconstruction situations:

- A **class diagram** showing the static structure of the program. In practice, such a diagram is extracted from the source code, either manually or with tools like Sniff+. In practice only some of the classes will have meaningful names. In our experiment this is mimicked by giving about half the classes meaningful names that reflect their roles in the application. In the class-diagram the classes are positioned such that the number of edge crossings in minimised, mimicking the application of an edge-crossing minimization algorithm to the extracted diagram.
- An (incomplete) description of the **dynamic behaviour**. In practice, the architects would extract this information from the source code. Because of time limitations this is explained to them verbally in our experiment.
- **Answers to questions** of the architects on specific details about the application were also given verbally. This mimics the iterative process [Demeyer et al, 1998] described, where the reconstructor obtains information about the software by analysing how specific aspects are handled.

Obviously the placement of the classes in the diagram can influence how the architects decompose the application. Using a different diagram for each architect with randomly placed classes could circumvent this. However, with these diagrams it would be much harder to explain the dynamic behaviour to the architects. Due to time limitations this was not considered possible. However, the architects were told explicitly that they should not let the placement of the classes influence their decisions. While creating their decomposition several architects made small tree-like diagrams to visualise possible decompositions that had a completely different structure than the provided class diagram. So although it is possible that

the class diagram's structure influenced the architects, we feel that the chosen approach sufficiently matches a practical architecture reconstruction case.

Figure 39 shows the class diagram of the simple-blackboard application. The diagram uses the static-structure notation of UML [Booch et al, 1999]. To simplify the diagram some dependencies have been omitted. More precisely, dependencies from class c_a to c_b for which:

- an association from c_a to c_b exists or
- c_a inherits from c_b

are omitted from the diagram, as [Booch et al, 1999] suggested.

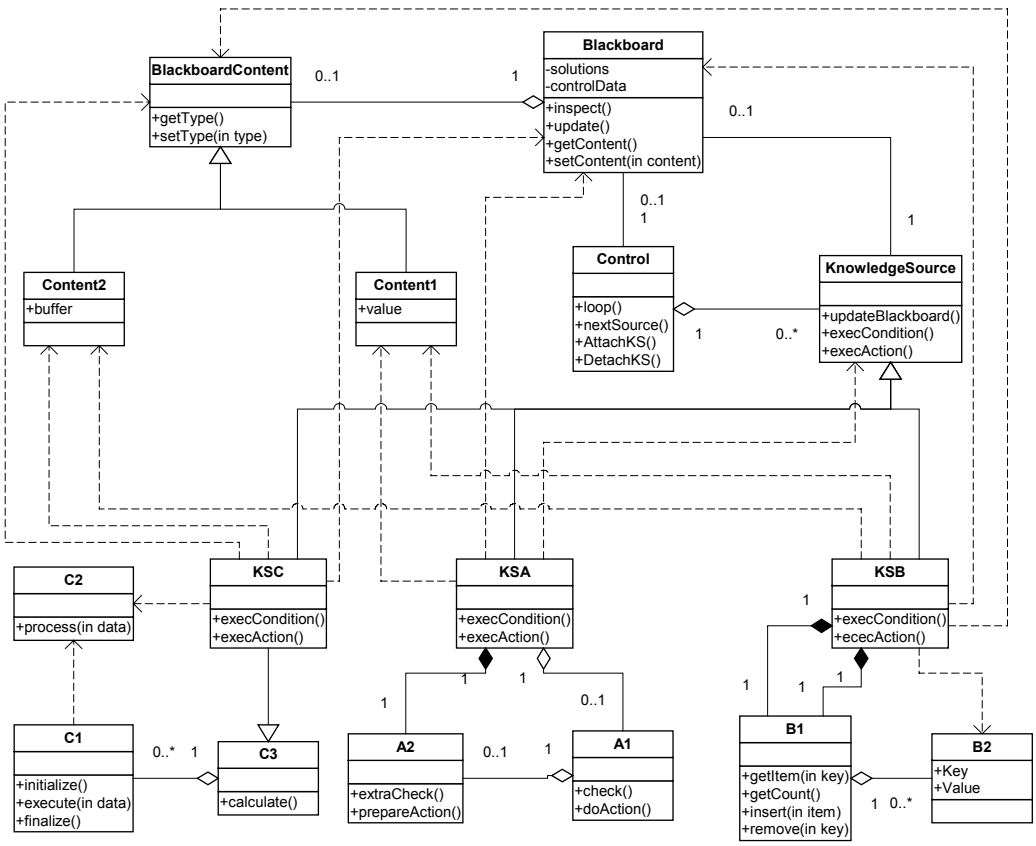


Figure 39: Simple-blackboard application class diagram

The application implements a simple data processing application that has been designed with extensibility in mind. Because the future interactions among the processing units are hard to predict, the architecture is based on the blackboard architectural style [Buschmann et al, 1999]. Some of the architects were not very familiar with this style. In those cases it was explained to them first. The application has three data processing classes, KSA, KSB and KSC. KSA reads a file from disk and places its content on the blackboard in a Content1 class. KSA uses two other classes, A1 and A2. KSB is triggered by the presence of Content1 data on the blackboard and uses this data as input for a set of complicated calculations. The results of these are placed on the blackboard in a Content2 class. KSB uses two other classes, B1 and B2 to perform the calculations. KSC is triggered by the presence of the Content2 class on the blackboard. This data is used as input for a complicated calculation, a part of which is inherited from the C3 class (reuse through inheritance). The results of this calculation are written to disk by KSC.

To prevent the triviality of the application from influencing the decomposition, the architects were told that each class represents a significant size, and that the diagram shows only public properties and methods, not the private ones. Further, they were told that they would not have to worry about limitations with respect to the number of human resources available.

Surprisingly, all architects produced a different decomposition. These are described in Appendix 5. To our knowledge no work has been published on the merging of a set of architectural decompositions. We have experimented with an approach that assigns an "attractive force" to each pair of classes that specifies their affinity. These forces are based on the decompositions produced by the architects. If an architect places two classes in the same subsystem this creates an attractive force between them. The decompositions are combined by summing the total forces for each pair of classes in the set of decompositions. In our experiments the dominant (i.e. strongest) forces determine the final combined decomposition, but a mechanism based on spring-layouts might be preferable. If architects have created a hierarchical decomposition, the attractive force between two classes increases with the number of levels on which the two classes are placed in the same subsystem. Because of time limitations we were not able to investigate the combining of decompositions further and decided to leave this as future work.

Besides the ten decompositions, Appendix 5 also describes the reasons that led to them. The majority of the architects based their decomposition on functional criteria. The architects that did consider the class-relations all considered dependencies to be the least important. The large number of dependencies in Figure 39 probably caused this. Inheritance was considered an important, but not compulsory indicator for a subsystem boundary. For the clustering parameters this suggests that the weight of inheritance relations (p_{w_g}) should be relatively low. The architects attempted to keep associations, especially the compositions, within a single subsystem. For the clustering process this suggests that the associations relations should have a relatively high weight (p_{w_a}).

Because of time limitations, and because an expert decomposition of the Océ Controller needs to be constructed anyway to assess the quality of the clustering result, it was decided not to use the simple-blackboard application to find a set of near-optimal parameters for the clustering process. Instead, two subsystems of the Océ Controller are used for this purpose. The question of how to merge a set of decompositions is left as future work.

Grizzly & Rip Worker

Instead of the simple-blackboard application, the Grizzly and RIP Worker subsystems described in paragraph 1.2 are used to determine a set of near-optimal parameter-tuples. These are then used to cluster the complete Océ Controller.
The expert decompositions of the Grizzly and RIP Worker subsystems are constructed with the two-step process described in paragraph 7.2.2. First, the "source-tree based clustering" module is used to create an approximation of the expert decomposition. Next, the architecture and design documents, together with information from the original developers, are used to refine this decomposition. The input of the original developers was especially valuable to classify classes in the implementation that are not mentioned in the design documentation.

Because it is not possible to enumerate all possible parameter-tuples, a subset must be selected. Recall that three of the five parameters are numeric and two of them are booleans. For each of the numeric parameters the search space is initially set to {0,1,2,3,4,5,6}. This gives a total of $7\times7\times7\times2\times2=1372$ combinations to investigate, each requiring ten executions of the clustering algorithm. Because combinations containing zero for all three numeric parameters do not export any information to the MDG-file these are not used. This leaves 1368 combinations to test.

To speed up the calculations AutoIt [AutoIt, 2005] scripts have been used to automate the clustering process. For every different parameter-tuple these scripts use the Bunch Exporter module to create an MDG-file. Next they repeat the following four steps ten times:

1. Cluster the MDG-file with Bunch.
2. Write the decomposition to the database with the Bunch Importer module.
3. Calculate the MoJoQuality and EdgeMoJo values with the MoJo module. This module writes these values to a file, which is used to determine the quality of the clustering for each parameter-tuple.
4. Remove the decomposition written in step 2 from the database.

In the remainder of this chapter the cycle of exporting the MDG-file once and the tenfold execution of these four steps is called a *ten-clusterings cycle*.

Grizzly and the RIP Worker have been processed separately, both on the platform described in Table 27. For Grizzly the ten-clusterings cycle took about four minutes, resulting in a total execution time of about 5488 minutes (91 hours) to test the 1372 combinations. For the RIP Worker this was about 3:20 (m:ss), resulting in a total execution time of about 4573 minutes (76 hours).

Table 20 shows the five best and five worst parameter-tuples for Grizzly and the RIP Worker, and the resulting EdgeMoJo and MoJoQuality. Observe that the two sets of best parameter-tuples are disjoint. For Grizzly the EdgeMoJo metric varies between 101,7 for the best and 169,7 for the worst decomposition. The MoJoQuality varies between 69,0% for the best and 57,3% for the worst decomposition. For the RIP Worker these figures are 42,6 and 67,9, and 66,3% and 55,0% respectively. These figures indicate that the choice of the clustering parameters affects the quality of the clustering result significantly.

Grizzly						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
2	5	5	false	false	101,7	69,0%
1	3	2	false	false	102,0	69,0%
0	0	6	true	false	102,2	69,8%
2	3	5	false	true	102,3	68,2%
1	4	3	false	false	102,7	68,9%
1358 other measurements						
4	0	0	false	false	169,1	61,2%
1	0	0	true	true	169,4	61,2%
3	0	0	false	true	169,4	61,2%
6	0	0	false	true	169,6	61,2%
0	3	0	false	false	169,7	57,3%

RIP Worker						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	5	2	false	false	42,6	66,3%
2	4	3	false	true	42,6	66,4%
1	6	4	true	true	42,7	66,7%
1	5	2	false	true	42,7	67,1%
2	4	1	true	false	42,7	66,0%
1358 other measurements						
1	0	0	true	false	67,4	55,0%
5	0	0	false	true	67,6	54,8%
3	0	0	true	false	67,8	55,1%
3	0	0	true	true	67,8	54,8%
4	0	0	true	false	67,9	55,0%

Table 20: Best five parameter-tuples for Grizzly and the RIP Worker

Appendix 6 shows the fifty "best" parameter-tuples and the produced clustering result for Grizzly and the RIP Worker. The tuples are sorted ascending according to the EdgeMoJo value. Note that, even though a large number of tuples are shown, this does not necessarily include the optimal parameter-tuple (assuming such a tuple exists). The reason for this is that only a subset of the search space has been investigated.

For the tuples that lead to a good clustering it is difficult to distinguish trends. The presence of parameter-tuples with zero for the numeric parameters indicates which types of relationships are important for the clustering result and which are not. When considering the best fifty parameter-tuples for Grizzly, tuples that have $p_{w_a}=0$ also have $p_{w_g}=0$. For the RIP Worker several tuples with $p_{w_a}=0$, but none with $p_{w_g}=0$ are present in the top fifty. So in both cases no tuples with $p_{w_g}=0$ and $p_{w_a}\neq 0$ are present in the top fifty. This indicates that ignoring the generalizations while taking the associations into account does not lead to a good clustering result. In other words, if the associations are used the generalizations must be used too.

With respect to the two boolean parameters (p_c and p_i) no trends can be distinguished. In the best fifty parameter-tuples all four possible combinations are represented equally.

For the parameter-tuples that lead to a poor quality clustering a clear trend is visible; for both Grizzly and the RIP Worker, the parameter-tuples with $p_{w_d}=0$ give the worst clustering result. Any parameter-tuple with $p_{w_d}\neq 0$ gives a better clustering result than the same parameter-tuple with $p_{w_d}=0$. This means that ignoring the dependencies leads to a poor quality clustering. Recall that the architects consulted for the reconstruction of the architecture of the simple-blackboard application considered dependencies to be the least important

architectural-indicator. Instead, most of them based their decomposition on functional criteria. The unexpected importance of dependencies for the clustering result can be explained in two ways:

1. Ignoring the dependencies leaves many classes without any connection to other classes (*unconnected classes*). These classes are then placed in the "unconnected classes" subsystem, which is probably not the right choice.
2. The presence and number of dependencies reflects the functional relations between the classes better than the associations and generalizations do.

If the first explanation holds, ignoring the dependencies must increase the number of unconnected classes much more than ignoring the other relationship-types. Table 21 shows the number of unconnected classes and the number of classes that are only connected with relations of the tree types, both for Grizzly and the RIP Worker. Observe that the number of unconnected classes in both subsystems is about the same. In the RIP Worker the number of classes that are only connected with dependency relations is relatively high compared to the other relationship types. But for Grizzly this is not the case; much more classes are only connected with a generalization than with a dependency. This means that the first explanation for the importance of the dependencies for the clustering result does not hold.
We therefore assume that dependencies are so important for the clustering result because they reflect the functional relations between the classes better than the other relationship-types.

Type of classes	Grizzly	RIP Worker
Unconnected	6	7
Only connected with dependency	4	19
Only connected with association	4	1
Only connected with generalization	9	3

Table 21: Connectivity of classes in Grizzly and the RIP Worker

Because no single best parameter-tuple could be identified it was decided to use a set of tuples instead of a single one. Recall that three different types of clusterings are performed; of one version only, and a class-relations-intersection and class-relations-union of two versions. Of these, the last produces the largest module dependency graph and therefore takes the longest to cluster. As can be seen in Table 25, this clustering takes almost one hour. Since each parameter-tuple leads to ten clusterings, we decided to test forty parameter-tuples on the Océ Controller.
The overlap between the set of best tuples for Grizzly and for the RIP Worker is very small. In fact, the set of twenty tuples that lead to the best clustering result for each of them are disjoint. We therefore decided to use the union of these two sets, leading to forty tuples to test.

7.4 Results of architectural-clustering case study

Now that a sub-optimal set of parameter-tuples has been identified the Océ Controller is clustered. The same procedure as described above is followed, where for every parameter-tuple ten clusterings are generated and the average quality is determined. To avoid basing conclusions on a single case, the architectures of the last two versions of the Océ Controller, 7d and 8a, are reconstructed.

The paragraphs below describe the results of these experiments. Paragraph 7.4.1 describes the results when clustering a module dependency graph that is based on a single version. Paragraph 7.4.2 describes the results when information from multiple versions is used.

7.4.1 Result when clustering one version

Table 22 shows the five parameter-tuples that, according to the EdgeMoJo metric, produced the best clusterings for version 7e (left) and 8a (right) of the Océ Controller. Recall that we use the EdgeMoJo metric to compare the clustering result to the result of a manual architecture reconstruction. So the parameter-tuples in Table 22 are those that produce

decompositions that come closest to a manually reconstructed architecture. Table 32 and Table 33 in Appendix 7 show the results for all forty tested parameter-tuples.

Version 7e						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
4	6	1	true	true	1.639,4	60,5%
1	4	1	true	true	1.644,9	60,5%
1	5	5	true	true	1.646,1	60,2%
1	1	5	true	false	1.646,8	60,3%
2	5	5	false	false	1.648,4	60,4%

Version 8a						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
4	6	1	true	true	1.477,1	62,5%
2	1	2	true	false	1.481,1	62,3%
6	3	4	true	false	1.481,3	62,5%
1	4	3	false	false	1.483,8	62,4%
2	3	5	false	true	1.484,2	62,3%

Table 22: Best five clusterings for version 7e and 8a of the Océ Controller

As shown in Table 22 a MoJoQuality of 60,5% was achieved for version 7e of the Océ Controller. The best clustering for version 8a had a MoJoQuality of 62,5%. Because in both cases the MoJoQuality for the best parameter-tuples exceeds 60% we consider these decompositions good starting points for manual refinement (see paragraph 7.1). Hence, these results confirm hypothesis H3. For both versions the parameter tuple (4,6,1,true,true) achieved the best clustering. Although it is tempting to conclude that this is the optimal parameter-tuple, this is probably a coincidence. Recall from Table 20 that for Grizzly and the RIP Worker different parameter-tuples led to the best clustering result.

To confirm hypothesis H4 the EdgeMoJo metric is used. This means that the addition of information from older versions must produce decompositions with an EdgeMoJo value that is lower than 1.639,4 for version 7e and lower than 1.477,1 for version 8a.

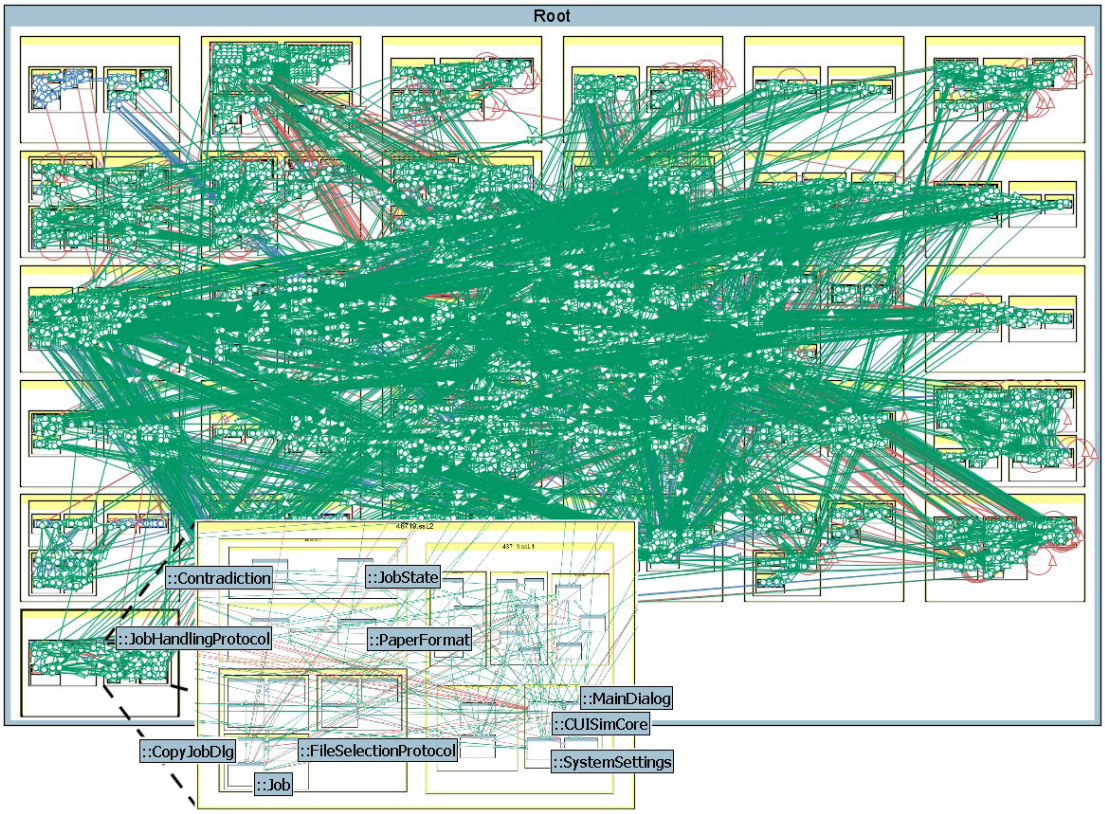


Figure 40: Example of a decomposition the clustering produced for version 8a

It is not possible to describe the produced decompositions completely in this thesis. To illustrate them, Figure 40 shows an example of a decomposition the clustering produced for version 8a with the parameter-tuple (4,6,1,true,true). This decomposition has a MoJoQuality

of 62% and an EdgeMoJo value of 1489. This means that this decomposition has approximately the same quality as the average shown in Table 22. In the view blue squares denote classes and yellow ones subsystems. The red, blue and green edges denote association, generalisation and dependency relations between classes respectively. Observe that the edges, especially the dependencies, obfuscate the view slightly. In Shrimp users can zoom in on subsystems to get a better view. In Figure 40 one subsystem is enlarged to mimic this, which is illustrated by the dotted lines. In the figure the names of several classes are shown in labels that refer to the class they are placed in. Labels that touch multiple classes refer to the class in the upper left corner.

Figure 41 shows the expert decomposition of version 8a. Observe that this decomposition contains fewer subsystems than the decomposition in Figure 40 but these are generally larger. The subsystem that matches the one zoomed-in on in Figure 40 is also enlarged here. Observe the differences between the two decompositions. For example ::Contradiction and ::JobState are placed in a separate subsystem.

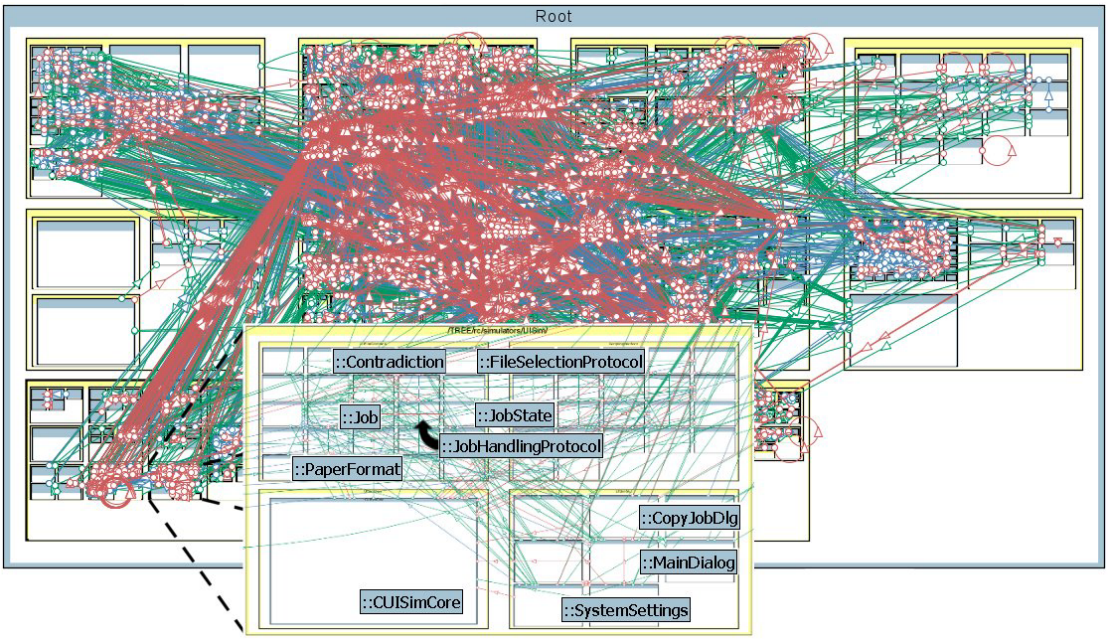


Figure 41: Expert decomposition of version 8a of the Océ Controller

7.4.2 Result when clustering with multiple versions

The “Combining version information” section in paragraph 7.2.1 describes the class-relations-intersection and class-relations-union operations for combining two models of different versions of a system. Recall from the same section that the internal structure of software is usually not decreasing monotonically. Refactorings that increase it again cause this. In order to prevent basing our conclusions on a single case we decided to combine the two reconstructed versions with *two* other versions, namely the first (version 1), and the version released before the one of which the architecture is reconstructed. This leads to four different version combinations for both the class-relations-union and the class-relations-intersection. We leave the testing of other combinations as future work.

Class-relations-intersection

Class-relations-intersection with first version

Table 23 shows the five parameter-tuples that produced the best clusterings for the class-relations-intersection of the two versions, 7e and 8a, with version 1. Table 34 and Table 35 in Appendix 7 show the results for all forty parameter-tuples.

Version 7e with 1						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	0	2	false	false	1.223,3	73,7%
0	0	1	true	false	1.229,5	73,7%
0	0	6	true	false	1.266,1	73,3%
1	5	6	false	true	1.286,6	72,3%
1	5	5	true	true	1.293,5	72,0%

Version 8a with 1						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
0	0	6	true	false	950,5	78,1%
0	0	1	true	false	958,0	78,1%
0	0	2	false	false	980,9	77,8%
0	1	1	true	false	1.006,3	76,8%
1	6	4	true	true	1.006,7	76,7%

Table 23: Best five clusterings for the class-relations-intersection with version 1

The class-relations-intersection of version 7e and 1 produced decompositions with an EdgeMoJo between 1223,3 and 1398,0. The MoJoQuality varied between 73,7% and 70,9%. For version 8a the EdgeMoJo varied between 950,5 and 1126,0, and the MoJoQuality between 78,1% and 75,5%.
Compared to the clustering based on the versions alone this is a significant quality improvement. The best tuple with the class-relations-intersection of version 7e and 1 has an EdgeMoJo value that is 25% lower than the best tuple when clustering 7e alone (from 1.639,4 to 1223,3). For version 8a the EdgeMoJo improved with 36% (from 1477,1 to 950,5).
For the parameter-tuples that lead to the worst clustering result, the class-relations-intersection with version 1 leads to a similar quality improvement, namely of 21% and 32% for version 7e and 8a respectively.

From this we conclude that basing the clustering on the class-relations-intersection with version 1 leads to a significantly better clustering. This confirms hypothesis H4 (see page 80).

Class-relations-intersection with previous version

Table 24 shows the five parameter-tuples that produced the best clusterings for the class-relations-intersection of the two versions, 7e and 8a, with the version released before them (7d and 7e respectively). Table 36 and Table 37 in Appendix 7 show the results for all forty parameter-tuples.

Version 7e with 7d						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
6	2	1	true	true	1.642,6	60,2%
1	5	5	true	true	1.644,8	60,3%
3	3	4	true	true	1.646,9	60,4%
1	5	2	false	true	1.651,3	60,3%
4	4	3	false	false	1.652,0	60,1%

Version 8a with 7e						
p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
6	1	6	false	true	1.642,6	59,1%
4	6	2	false	true	1.644,7	59,0%
2	4	3	false	true	1.649,1	59,1%
6	5	4	false	true	1.651,2	59,0%
3	3	1	false	false	1.652,3	58,9%

Table 24: Best five clusterings for the class-relations-intersection with the previous version

The class-relations-intersection of version 7e and 7d produced decompositions with an EdgeMoJo between 1.642,6 and 1.804,1. The MoJoQuality varied between 60,2% and 57,4%. For version 8a the EdgeMoJo varied between 1.642,6 and 1.811,8, and the MoJoQuality between 59,1% and 56,0%.
These results are similar to the results achieved when using only information from the version of which the architecture is reconstructed. Recall that in that case for version 7e the EdgeMoJo value was 1.639,4 to 1.778,7, which is similar to the result achieved here. Clustering version 8a with only information from that version achieves decompositions with an EdgeMoJo value between 1.477,1 and 1.661,4, which is slightly better than the result achieved here. The MoJoQuality metric shows the same pattern. This means that basing the clustering on the class-relations-intersection with the previous version does not lead to a better clustering result.

Class-relations-union

Class-relations-union with first version

Table 25 shows the clustering results for the five best clusterings of the class-relations-union of version 8a and 1. Table 38 in Appendix 7 shows all forty tested parameter-tuples.

p_{w_a}	p_{w_g}	p_{w_d}	P_c	p_i	Edge MoJo	MoJo Quality
3	3	4	true	true	1.458,9	62,9%
4	4	3	false	false	1.459,5	62,9%
3	6	1	false	false	1.461,2	62,7%
1	4	3	false	false	1.461,6	62,8%
6	5	4	false	true	1.462,3	62,7%

Table 25: Best five clusterings for the class-relations-union of version 8a and 1

The clustering of the class-relations-union of version 8a and 1 achieved an EdgeMoJo value between 1458,9 and 1619,8. In the case where version 8a was clustered alone the EdgeMoJo was between 1468,3 and 1661,4. This means that the class-relations-union does not lead to an improvement of the quality of the clustering. The MoJoQuality for the class-relations-union lies between 62,9% and 59,9%, which is also comparable to the result achieved when clustering with version 8a alone (62,6% to 59,3%).

Class-relations-union with previous version

Table 26 shows the clustering results for the five best clusterings of the class-relations-union of version 8a and 7e. Table 39 in Appendix 7 shows all forty tested parameter-tuples.

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	Edge MoJo	MoJo Quality
4	4	3	false	false	1.458,5	62,7%
6	3	4	true	false	1.466,8	62,6%
4	6	1	true	true	1.468,5	62,7%
6	1	6	false	false	1.468,6	62,5%
3	6	1	false	false	1.469,4	62,6%

Table 26: Best five clusterings for the class-relations-union of version 8a and 7e

The clustering of the class-relations-union of version 8a and 7e achieved an EdgeMoJo value between 1458,5 and 1622,2. Similar to the class-relations-union with version 1, this is comparable to the results when clustering version 8a alone (EdgeMoJo between 1468,3 and 1661,4). The MoJoQuality metric confirms this. It now has a value between 62,7% and 59,8%, which is similar to the value achieved when clustering with version 8a alone (62,6% to 59,3%).

This leads to the conclusion that combining two versions with the class-relations-union operator does not lead to an improvement of the clustering result. We therefore decided not to test other combinations of versions.

7.4.3 Observations

The quality of the decompositions our architectural clustering method produced is relatively good in the sense that they approach the result of a manual architecture reconstruction relatively well. In our experiments where the architecture of two versions of the Océ Controller was reconstructed the produced decompositions had a MoJoQuality of 60,5% and 62,5% respectively. This exceeds the goal of 60% set in paragraph 7.1 and hence confirms hypothesis H3.

Incorporating information from other versions in the clustering process improved the clustering result in some cases:

- Class-relations-intersection:
    - With the first version: improvement of about 20% to 35%.
    - With the previous version: no improvement.
- Class-relations-union:
    - With the first version: no improvement.
    - With the previous version: no improvement.

This confirms hypothesis H4.

The architectural-clustering workbench uses Sniff+ to extract facts from the source code. Sniff+ proved to be a reliable and stable fact extractor. The Sniff API caused some problems but these could be circumvented in most cases. The only situation where this was not the case was with associations that were based on C++ templates. Because of the low number of these associations in the Océ Controller this had little impact on the clustering result however.

The workbench uses Bunch to cluster the classes that were extracted from the source code. To our knowledge Bunch has not been used to cluster object-oriented software before. Considering the quality of the decompositions our workbench produced we conclude that this is no problem.

In the experiment the expert decomposition was constructed with a two-step approach:

1. Reconstruct an approximation of the expert decomposition based on the structure of the source-tree.
2. Refine this approximation using architectural and design documentation, as well as information from the system's architects.

Recall that our approach assumes that an architect will manually refine the produced decomposition, which is comparable to step 2 above. This raises the question if our clustering method achieves better decompositions than the source-tree based architecture reconstruction. The MoJoQuality of the latter decomposition is 91%, which is better than the results our method achieved (best MoJoQuality was 78%). This indicates that the structure of the source-tree provides valuable information for architecture reconstruction.
The ACDC clustering algorithm, which is described in paragraph 6.2.3, also uses information from the source-tree in the clustering process. It clusters procedural code, using information of relations between source code entities (e.g. files or procedures) and of the source-tree itself. However, its decompositions have a MoJoQuality of around 60%, which is comparable to the quality of the results our method achieved. The high quality of the decomposition created from the source-tree of the Océ Controller can be explained in two ways; either the algorithm we use to create a decomposition from the source-tree is much better than ACDC, or the source-tree of the Océ Controller reflects the architecture relatively well. We speculate that the latter is the case. Nevertheless, incorporating source-tree information in the clustering process seems to have the potential to improve the quality of the clustering result. Due to time limitations we cannot explore this further and have to leave this as future work.

Execution times

All performance figures described in this chapter are measured on the test platform of which the characteristics are shown in Table 27.

Processor	Pentium 4; 2,0 GHz
Memory	2 GB
Operating system	Windows 2000 SP4
Java	1.4.2_06
Sniff+	4.2 CP2
MySQL	4.1.8-nt
Bunch	3.3.6
Shrimp	2.0 build 2
Rigi	6.0, version 2-Oct-2003

Table 27: Test system characteristics

To speed up the measurements, a second PC has been used for some calculations. During the clustering of the Océ Controller for example, each of the two available PCs processed about half the set of parameter-tuples. This second PC has the same characteristics as listed in Table 27, but with a 2.8 GHz Pentium 4 processor instead.

The execution times to cluster Grizzly and the RIP Worker are described in the "Grizzly & Rip Worker" section in paragraph 7.3.2. Table 28 shows some representative examples of the time needed to execute the essential steps of the architectural clustering process for the Océ Controller. All values are measured in wall-clock time.

Observe in Table 28 that the fact extraction and subsequent Sniff Import take a lot of time. This confirms our assumption in paragraph 7.2.2 that led to the use of a database to store the extracted facts. About half of this time is spent creating Sniff's internal meta-data repository and parsing the source code. The other half is spent importing this information in the database. Both need to be done only once for every analysed version. Note that building the complete Océ Controller from source code takes about one to two hours on our test platform, which is about one order less.
Note also the times needed to cluster version 8a alone, and the class-relations-intersection and -union (0:18, 0:11 and 0:58 respectively). These values are approximately proportional to the size of the respective module dependency graphs.

Task	**Time (hh:mm)**
Fact extraction of version 8a (Sniff+ parsing and import in database)	21:19
Clustering version 8a (Bunch Export, Clustering and Bunch Import)	0:18
Clustering class-relations-intersection of version 8a and 1 (Bunch Export, Clustering and Bunch Import)	0:11
Clustering class-relations-union of version 8a and 1 (Bunch Export, Clustering and Bunch Import)	0:58
MoJo calculations for version 8a	0:05
Visualization version 8a (RSF Export and loading in Shrimp)	0:05

Table 28: Execution times for the Océ Controller (wall-clock time)

The ten-clusterings cycle described in paragraph 7.3.2 combines several of these steps. Table 29 shows the execution times of the executed ten-clusterings cycles. The class-relations-union of version 7e with the first and the last version have not been measured. These times are marked "n.m."

Task	**Time (hh:mm)**	
	7e	**8a**
One version	2:58	3:11
Class-relations-intersection with first version	1:01	1:02
Class-relations-intersection with previous version	3:03	2:42
Class-relations-union with first version	n.m.	4:41
Class-relations-union with previous version	n.m.	3:34

Table 29: Execution times of the ten-clusterings cycles (wall-clock time)

Problems encountered

The previous paragraphs mention several problems we encountered during this case study:

- The Sniff API does not export association relations defined with template-based variables. Further, the Sniff API exports namespaces with or without the parent namespaces in an unpredictable manner. Both issues are described in the "Sniff and Sniff Import" section in paragraph 7.3.1.

- The models extracted from the source code of the Océ Controller contain classes that are not involved in any of the extracted relations. Because they are not connected to other classes, the clustering algorithm cannot classify them. The workbench handles this by placing these classes in a special subsystem for “unconnected classes”, as is described in the “Bunch Export, Bunch & Bunch Import” section paragraph 7.2.2.
- The source-tree of the Océ Controller contains multiple classes with the same name and namespace. Examples are stubs, test tools and experimental versions of subsystems. In cases where no other identifiers are available to identify the classes this causes problems. This is handled by using the class-name and source-file to identify classes, as is described in the “Sniff+ & Sniff Import” section in paragraph 7.2.2.

The size of the Océ Controller caused several problems. First of all, MySQL could only execute the queries after tuning it for large databases. Second, a complete cycle of fact extraction, clustering and result assessment or visualization took a significant amount of time. For one clustering cycle this is no problem, but when a large number of clustering cycles are performed it is. Effectively, this limits the number of different clusterings that can be created, and hence the number of different parameter-tuples that can be tested. Note that in practical clustering-based architecture reconstruction cases only one, or a limited number of clusterings are generated. Therefore this limitation applies mainly to projects experimenting with different clustering approaches or parameters, and not to practical architecture reconstruction cases.

7.5 Conclusions of the architectural-clustering case study

This case study aimed to investigate the following hypotheses:

H3: Automatic clustering-based architecture reconstruction methods can reconstruct an architectural view of the Océ Controller from its source code that is a good starting-point for manual refinement.

H4: Utilizing information obtained from source code of older versions can improve the quality of the output of architectural clustering algorithms for more recent versions of a system.

Paragraph 7.1 quantifies a decomposition architectural clustering produced as good if it has a MoJoQuality of at least 60% relative to the result of a manual architecture reconstruction.

To confirm hypothesis H3 and H4 an architecture reconstruction workbench has been constructed that implements clustering-based architecture reconstruction. The implemented approach is based on information that is always available from object-oriented source code and does not assume the availability of any other information.
The architecture of two versions of the Océ Controller has been reconstructed with this workbench. The resulting decompositions have been compared to expert decompositions to compare their quality.

This leads to the following conclusions:
- Architectural clustering based on structural relations between the classes can reconstruct architectural views of object-oriented software that are useful for software maintenance.
- The execution time is such that clustering the complete Océ Controller is feasible in practice.
- Sniff+ can be used to extract facts from the source code of the Océ Controller with reasonable reliability and accuracy.
- Bunch can also cluster object-oriented software, and not just procedural software. Further, Bunch can handle the use of different weights for different types of relations instead of using the same weight for all relations.
- The weight of the relationship-types significantly affects the quality of the clustering result. However, in our experiments there was no single combination of weights that produced the best clusterings for all analysed pieces of software.

- Dependency relations are very important for the quality of the clustering result. In all experiments ignoring the dependencies led to a reduction of the clustering result's quality, regardless of the weight assigned to the other relationship-types.
- The quality of the clustering result improves if the clustering is based on those class-relations that are *also* present in the first version of the software (class-relations-intersection with version one). If instead of the first the previous version is used no improvement is achieved. This might be due to the fact that in the previous version the architecture is deteriorated much further than in the first version.
- Architectural clustering based on the class-relations present in the clustered version *or* the first one (class-relations-union) does not lead to better clustering results. The same holds for the combination with the previous version instead of the first.

From this case study we conclude that architectural clustering reconstructs an architecture from the source code of the Océ Controller that is a good starting point for manual refinement. In this refinement some small adjustments need to be made, such as the moving of some classes to another subsystem. This confirms hypothesis H3. We further conclude that basing the clustering on the class-relations that are also present in the first version of the system leads to a better clustering result. This confirms hypothesis H4.

8 Conclusions and future work

8.1 Conclusions

This thesis started with the following research questions:
1. Which methods are available for architecture reconstruction?
2. Can these methods be used to reconstruct the architecture of the Océ Controller?
3. How good are the results?
4. How can these methods be improved?

In literature several methods for architecture reconstruction are described, including manual reconstruction, pattern detection, architectural clustering and architectural slicing. For large software systems completely manual methods are not practical. Of the automatic methods, pattern detection and architectural clustering are the most prominent ones. We have applied these two methods to the Océ Controller in two case studies.

8.1.1 Pattern detection

In literature pattern detection methods that are based on a pattern library have been applied frequently and their properties are relatively well known. A disadvantage is that they require upfront knowledge on the used patterns and their precise implementation. Implementation variations make the latter difficult to specify. The pattern detection method we applied is based on mathematical Formal Concept Analysis and does not require a pattern library. It detects structural patterns in two subsystems of the Océ Controller.

The method proved to be able to detect frequently used design structures in source code. However, even the detection of relatively simple structures in relatively small pieces of source code required a lot of calculations. Since this is inherent to the used algorithms, the application of this technique to reconstruct architectural views of large object-oriented systems, and more specific the Océ Controller, is not considered practical. It is possible to detect design patterns in its subsystems though. These have a size of about five to ten percent of the complete system. Note that the method detects structural constellations of classes, and not named design patterns such as those [Gamma et al, 1995] described.

Besides performance issues, the reduction of the large number of similar patterns in the output is also important. Based on the complexity of the patterns we filtered the output, but the results show the more advanced filtering is necessary in order for the method to be useful.

8.1.2 Architectural clustering

Architectural clustering uses mathematical clustering techniques to group closely related source code elements into suitable higher-level abstractions. In the context of architecture reconstruction it has mainly been applied to procedural code and only occasionally to object-oriented software like the Océ Controller.

In several experiments we have applied architectural clustering to reconstruct the architecture of two versions of the Océ Controller, using the Bunch tool. To our knowledge this tool has not yet been previously used to cluster object-oriented software. The experiments show that it is possible to reconstruct architectural views of large object-oriented software systems such as the Océ Controller with architectural clustering that are useful for software maintenance. Based on experiences reported in literature for procedural code we expected that these views required some manual refinement such as the relocation of some classes. Our experiments confirmed this, but also showed that the reconstructed views are good starting points for this process.

In our experiments the clustering groups classes based on the structural relations between them. We distinguished three types of relations; associations, generalizations and dependencies. To our knowledge no research has been performed on how these types should be converted into the graph that is clustered, and which types are most important for the quality of the clustering result. We have experimented with several combinations of

relations, assigning different weights to each type. These experiments showed that the weights have a significant effect on the quality of the clustering result. However, for each of the analysed pieces of software a different combination of weights produced the clustering that came closest to a manually reconstructed architecture. Despite this, all our experiments clearly showed that the dependencies are more important for the quality of the clustering result than associations and generalizations.

The execution time of the clustering process is reasonable, but still substantial for the Océ Controller. On our test system it took about five to ten minutes. Extracting the structural information from the source code on the other hand takes a lot of time. On our test system it took over twenty *hours*. This is due to the creation of Sniff's meta-data repository, parsing the sources and transferring this information to the workbench's database. Usually this is only done once. If multiple fact extractions are necessary incremental fact extraction could reduce the total time significantly. Note that this is a fully automatic process that does not require user-interaction.

If multiple versions of a system have been released the clustering process can be based on information from multiple versions. To our knowledge architectural clustering case studies thus far only used information from a single version. We have experimented with several ways to incorporate information from multiple versions in the clustering process. In these experiments we observed the following behaviour:

- In cases where the clustering was only based on the structural relations that were *also* present in the first version the clustering result had a better quality compared to when only information from the reconstructed version was used.
- Basing the clustering on the relations that are present in the reconstructed version *and* the version released before it did not lead to a better clustering result. This might be due to the fact that in the previous version the architecture has deteriorated much more than in the first version.
- The clustering can also be based on the relations that are present in the reconstructed version *or* another version. However, neither the combination with the first version, nor the combination with the previous version led to an improved clustering result compared to the result obtained when using only information from the reconstructed version.

8.1.3 Concluding remarks

We therefore conclude that pattern detection without a pattern library is not practical for the complete Océ Controller. Architectural clustering on the other hand appears to be a useful technique for reconstructing architectural views of this system. Despite that this technique works completely automatic, manual refinement of the results is still needed though.

8.2 Future work

8.2.1 Pattern detection

In the pattern detection case study described in chapter 5 many patterns have been found that are highly similar to each other. This makes it difficult to use this information for program understanding. In paragraph 5.2.4 we have described several filters to remove uninteresting patterns from the output, but these are not sufficient. If the method is used in practice better filtering is required. It might also be possible to group similar patterns into groups and show a single pattern of each group to the user. The similarity of patterns could be based on the number of edges that must be added and removed to transform them into each other, as is suggested in the "Quality of the results" section in paragraph 5.4.3.

Finding frequently used design constructs in the source code essentially finds frequently occurring subgraphs in the class graph. An alternative to the pattern detection used in chapter 5 might be to use graph compression algorithms that are based on the detection of recurring subgraphs. We have built a small prototype that uses the Subdue algorithm [Jonyer et al, 2001]. This algorithm creates a list of recurring subgraphs and replaces all occurrences of these subgraphs with references to this list. However, when this algorithm is used for pattern detection the fact that the algorithm looks for perfectly identical subgraphs causes problems.

The intertwining of structures often encountered in practice caused this prototype to find no patterns at all in two subsystems of the Océ Controller[36]. Lossy graph compression algorithms might introduce the required fuzziness, but due to time limitations we were not able to explore this further. Note that the FCA-based approach described in chapter 5 does not have this problem.

8.2.2 Architectural clustering

The clustering described in chapter 6 assigns names to subsystems that are based on the cluster-names Bunch generated. These names have little meaning to humans and should be replaced with meaningful names.

The clustering workbench reconstructs the architecture from scratch. Adapting the workbench such that the clustering starts from a user-specified state allows users to incorporate their knowledge of the architecture. This has the advantage that the workbench can more easily be applied to software of which the architecture is partially known.

Our architectural clustering uses several user-specified parameters that affect the information on which the clustering is based. As described in paragraph 7.3.2 and 7.4 our experiments show that these have a significant effect on the quality of the clustering result for several cases. It is not clear however which values *in general* achieve the best clustering results. We have explored many different values, but leave many others unexplored. Further research is required on this matter.

We experimented with combing structural information of multiple versions to improve the quality of the clustering result. Although several version-combinations have been tested, many more combinations are imaginable. Testing these other combinations might reveal combinations that lead to even better clusterings.

Our clustering approach is based on structural information about relations between classes. The approach does not use information about the structure of the source-tree in which the classes are defined. In the case of the Océ Controller the structure of the source tree seems to represent information that can be used to improve the quality of the clustering further, as is suggested in paragraph 7.4.3. Besides this, dynamic information from for example traces could also be beneficial.

In the “Simple-blackboard application” section in paragraph 7.3.2 an experiment is described that aimed to find an expert decomposition of a small existing program. In the experiment ten architects individually decomposed an architecture. Surprisingly all architects produced a different decomposition, giving a set of decompositions from which the expert decomposition needed to be derived. We have experimented with several methods to achieve this, but were not able to explore this thoroughly due to time limitations. The “Simple-blackboard application” section describes several ideas that seem worth investigating.

[36] This refers to Grizzly and the RIP Worker subsystems.

Appendix 1 References

[Alexander, 1979] C. Alexander. The Timeless Way of Building. Oxford University Press, New York, NY, 1997.

[Andritsos and Tzerpos, 2003] P. Andritsos, V. Tzerpos. Software Clustering based on Information Loss Minimization. In Proc. of the 10th Working Conference on Reverse Engineering (WCRE'03), Nov. 2003, p. 334.

[Anquetil and Lethbridge, 1999] N. Anquetil, T.C. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In Proc. of the Sixth Working Conference on Reverse Engineering (WCRE'99), 1999, p. 235.

[Antoniol et al, 1998] G. Antoniol, R. Fiutem, L. Cristoforetti. Design Pattern Recovery in Object-Oriented Software. In Proc. of the 6th International Workshop on Program Comprehension, 1998, p. 153.

[Arévalo, 2003] G. Arévalo. Understanding Behavioral Dependencies in Class Hierarchies using Concept Analysis. In Proc. of LMO 2003: Languages et Modeles `a Objets, Jan. 2003, pp. 47-59.

[Arévalo and Mens, 2002] G. Arévalo, T. Mens. Analysing Object-Oriented Application Frameworks Using Concept Analysis. In J-M. Bruel and Z. Bellahsène, editors, Advances in Object-Oriented Information Systems - OOIS 2002 Workshops, number 2426 in LNCS, Springer, 2002, pp. 53-63.

[Arévalo et al, 2003] G. Arévalo, S. Ducasse, O. Nierstrasz. Understanding classes using X-Ray views. In Proc. of 2nd International Workshop on MASPEGHI 2003 (ASE 2003), Oct. 2003, pp. 9-18.

[Armstrong and Trudeau, 1998a] M.N. Armstrong, C. Trudeau. Evaluating Architectural Extractors. In Proc. of the 1998 Working Conference on Reverse Engineering (WCRE'98), p. 30.

[Armstrong and Trudeau, 1998b] M.N. Armstrong, C. Trudeau. CS 746G Project: Evaluating Architecture Extraction Tools. Apr. 1998.

[AutoIt, 2005] AutoIt v3 Home Page – AutoIt3. Apr. 2005. http://www.hiddensoft.com/autoit3/

[Ball, 1999] T. Ball. The concept of Dynamic Analysis. In Foundations of Software Engineering, Proc. of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT int. symposium on Foundations of software engineering, 1999, p. 216.

[Bär et al, 1999] H. Bär, M. Bauer, O. Ciupke, S. Demeyer, S. Ducasse, M. Lanza, R. Marinescu, R. Nebbe, O. Nierstrasz, M. Przybilski, T. Richner, M. Rieger, C. Riva, A.M. Sassen, B. Schulz, P. Steyaert, S. Tichelaar, J. Weisbrod. The FAMOOS Object-Oriented Reengineering Handbook. Oct., 1999.

[Bass et al, 2003] L. Bass, P. Clements, R. Kazman. Software Architecture in Practice, second edition. Pearson Education Inc., Boston, MA, USA. 2003.

[Bassil and Keller, 2001] S. Bassil, R.K. Keller. Software Visualization Tools: Survey and Analysis. In Proc. of the 9th International Workshop on Program Comprehension (IWPC'01), 2001, p. 7.

[Bauer and Trifu, 2004] M. Bauer, M. Trifu. Architecture-Aware Adaptive Clustering of OO Systems. In Proc. of the eights European Conference on Software Maintenance and Reengineering (CSMR'04), 2004.

[Beck, 1997] K. Beck. Smalltalk Best Practice Patterns. Prentice Hall, 1997.

[Beck and Eichmann, 1993] J. Beck, D. Eichmann. Program and Interface Slicing for Reverse Engineering. In Proc. of the 15th International Conference on Software Engineering (ICSE'93), 1993, pp. 509-518.

[Beck et al, 1996] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, J. Vlissides. Industrial Experience with Design Patterns. In Proc. of the 18th International Conference on Software Engineering (ICSE-18), 1996, pp. 103-114.

[Bellay and Gall, 1997] B. Bellay, H. Gall, An Evaluation of Reverse Engineering Tools. Technical report TUV-1841-96-01, Technical University of Vienna, Mar. 1997.

[Bennett and Rajlich, 2000] K. H. Bennett, V.T. Rajlich. Software maintenance and evolution: a roadmap. In Proc. of the Conference on the Future of Software Engineering, June 2000, pp. 73-87.

[Berkhin, 2002] P. Berkhin. Survey of Clustering Data Mining Techniques. Technical report, Accrue Software, San Jose, California, 2002.

[Bischofberger, 1992] W.R. Bischofberger. Sniff-A Pragmatic Approach to a C++ Programming Environment. In Proc. of the USENIX C++ Conference, Aug. 1992

[Booch, 1994] G. Booch. Object-Oriented Analysis and Design with Applications. Benjamin/Cummins Publishing Company Inc, 2nd edition, 1994.

[Booch et al, 1999] G. Booch, J. Rumbaugh, I. Jackobson. The Unified Modeling Language User Guide. Addison Wesley Longman Inc, 1999.

[Bordat, 1986] J.P. Bordat. Calcul pratique du treillis de Galois d'une correspondance, Math. Sci. Hum., 96, 1986, pp. 31-47.

[Bowman and Holt, 1998] I.T. Bowman and R.C. Holt. Software Architecture Recovery Using Conway's Law. In Proc. of CASCON'98, 1998, pp. 123-133.

[Bowman et al, 1999] I.T. Bowman, R.C. Holt, N.V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In Proc. of the 21st Int. Conference on Software Engineering (ICSE'99), 1999, pp.: 555-563.

[Buckley, 1989] J. Buckley. Some standards for software maintenance. Standards, IEEE Computer, Nov. 1989.

[Bunch, 2005] Bunch homepage. Mar. 2005. http://serg.cs.drexel.edu/projects/bunch/

[Buschmann et al, 1999] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley and Sons Ltd, Chichester, England, Apr. 1999.

[Casais, 1998] E. Casais. Re-engineering object-oriented legacy systems. In Journal of Object-Oriented Programming (JOOP), 10(8), 1998, pp. 45-52.

[Chen et al, 1998] Y.R. Chen, E.R. Gansner, E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In IEEE Transactions on Software Engineering, 24(9), Sept. 1998, pp. 682-694.

[Chikovsky and Cross, 1990] E.J. Chikovsky, J.H. Cross. Reverse Engineering and Design Recovery: A taxonomy. In IEEE Software, 7(1), Jan. 1990, pp. 13-17.

[Choi and Scacchi, 1990] S.C. Choi, W. Scacchi. Extracting and Restructuring the Design of Large Systems. In IEEE Software, 7(1), Jan. 1990, pp. 66-71.

[Columbus, 2003] Setup and User's Guide to Columbus/CAN, Academic Version 3.5. FrontEndART Ltd, Jan. 2003.

[Conway, 1968] M.E. Conway. How Do Committees Invent? In Datamation Magazine, 14(4), Apr. 1968, pp. 28-31.

[Dekel, 2002] U. Dekel. Applications of concept lattices to code inspection and review. In The Israeli Workshop on Programming Languages and Development Environments, chapter 6. IBM Haifa Research Lab, IBM HRL, Haifa University, Israel, July 2002.

[Dekel and Gil] U. Dekel, J. Gil. Revealing Class Structure with Zoomable Concept Lattices. Technion – Israeli Institute of Technology, Department of Computer Science.

[Dekleva, 1992] S. Dekleva. The Influence of the Information Systems Development Approach on Maintenance. In MIS Quarterly, Sept. 1992, pp. 355-372.

[Delnooz and Vrijnsen, 2003] C. Delnooz, L.J.G. Vrijnsen. The Caribou Project: a scenario-based approach towards a prototyping framework. Graduation thesis, Technische Universiteit Eindhoven, Stan Ackermans Instituut, 2003.

[Demeyer et al, 1998] S. Demeyer, S. Tichelaar, and P. Steyaert. Definition of a Common Exchange Model. Technical report, University of Bern, July 1998.

[Demeyer et al, 2004] S. Demeyer, S. Ducasse, O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann Publishers, San Francisco, CA, USA. 2004.

[Deursen, 2001] A. van Deursen. Software Architecture Recovery and Modelling [WCRE 2001 Discussion Forum Report]. ACM SIGAPP Applied Computing Review, 10(1), 2002.

[Ding and Medvidovic, 2001] L. Ding, N. Medvidovic. Focus: A Light-Weight, Incremental Approach to Software Architecture Recovery and Evolution. In Proc. of the 2001 Working IEEE/IFIP Conference on Software Architecture (WISCA'01), 2001.

[Doval et al, 1999] D. Doval, S. Mancoridis, B.S. Mitchell. Automatic Clustering of Software Systems using a Genetic Algorithm. In Proc. of the Software Technology and Engineering Practice, 1999 (STEP '99). Feb. 1999, pp. 73-81.

[Ducasse, 2003] S. Ducasse. Reengineering Object-Oriented Applications. Insitut für Informatik und Angewandte Mathematik, University of Bern, Switzerland, Sept. 2003, IAM-03-008.

[Ducasse et al 2004] S. Ducasse, T. Girba, J.-M. Favre. Modeling Software evolution by Treating History as a First Class Entity. In Proc. of the Workshop on Software Evolution Through Transformations (SETra 2004).

[DVRIP, 2002] Design View of the RIP Worker. Océ Engineering Venlo: Product Document. Apr. 2002.

[Eisenbarth et al, 2001] T. Eisenbarth, R. Koschke, D. Simon. Aiding Program Comprehension by Static and Dynamic Feature Analysis. In Proc. of the International Conference on Software Maintenance, Nov. 2001, pp. 602-611.

[Ehrlich et al, 1990] W.K. Ehrlich, J.P. Stampfel, J.R. Wu. Application of software reliability modeling to product quality and test process. In Proc. of the 12th International Conference on Software Engineering, Mar. 1990, pp. 108-116.

[Erlikh, 2000] L. Erlikh. Leveraging legacy system dollars for E-business. IT Professional, May/June 2000, 2(3), pp. 17-23.

[Fabry and Mens, 2003] J. Fabry, T. Mens. Language-Independent Detection of Object-Oriented Design Patterns. In Proc. of the European Smalltalk User Group 2003, Aug. 2003.

[Ferenc et al, 2001] R. Ferenc, J. Gustafsson, L. Müller, J. Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. In Proc. of the 7th Symposium on Programming Languages and Software Tools (SPLST 2001), June 2001, pp. 58-70.

[Ferenc et al, 2004] R. Ferenc, Á. Beszédes and T. Gyimóthy. Extracting Facts with Columbus from C++ Code. In Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004), Mar. 2004, pp. 4-8.

[Finnigan et al, 1997] P.J. Finnigan, R.C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H.A. Muller, J. Mylopoulos, S.G. Perelgut, M. Stanley, K. Wong. The Software Bookshelf. In IBM Systems Journal, 36(4) (Nov. 1997), pp. 564-593.

[Fjeldstadt and Hamlen, 1984] R.K. Fjeldstadt, W.T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. Proc. GUIDE 48, IEEE Computer Society Press, Apr. 1984.

[Foote and Yooder, 2000] B. Foote. and J. Yoder. Big ball of mud. In N. Harrison, B. Foote and H. Rohnert, editors, Pattern Languages of Program Design 4, Addison-Wesley, 2000, pp. 653-692.

[Fowler et al, 1999] M. Fowler, K. Beck. J. Brant. W. Opdyke, D. Roberts. Refactoring: Improving the design of Existing Code. Addison-Wesley, 1999.

[Gamma et al, 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: elements of reusable object-oriented software, fifth edition. Addison-Wesley, Dec. 1995.

[Galicia, 2005] Galicia Project. Feb. 2005, http://www.iro.umontreal.ca/~galicia/.

[Ganter, 1987] B. Ganter.Beiträge zur Begriffsanalyse, chapter Algorithmen zur Formalen Begriffsanalyse. BI-Wissenschaftsverlag, 1987.

[Ganter and Wille, 1998] B. Ganther, R. Wille. Applied lattice theory: formal concept analysis. In General Lattice Theory, G. Grätzer editor, Birkhäuser Verlag, 1998

[Gîrba and Lanza, 2004] T. Gîrba and M. Lanza. Visualizing and Characterizing the Evolution of Class Hierarchies. In Proc. of the Fifth International Workshop on Object-Oriented Reengineering (WOOR 2004), 2004.

[Godfrey and Lee, 2000] M.W. Godfrey and E.H.S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In Proc. of the Second International Symposium on Constructing Software Engineering Tools (CoSET00), June 2000, pp. 15-23.

[Graphviz, 2005] Graphviz - Graph Visualization Software. Feb. 2005, http://www.graphviz.org/.

[Grass, 1998] Robert Grass. Software Maintenance: Less Is Not More. In IEEE Software, July/Aug. 1998, pp. 67-68.

[Grizzly, 2003] Grizzly – Architecture Document. Océ Research Report, Aug. 2003.

[Guo et al, 1999] G.Y. Guo, J.M. Atlee and R. Kazman. A Software Architecture Reconstruction Method. In Proc. of the First Working IFIP Conference on Software Architecture (WICSA), Feb. 1999, pp 15-33.

[Hassan, 2002] A.E. Hassan. Architecture Recovery of Web Applications. University of Waterloo, Ontario, Canada, 2002.

[Hassan and Holt, 2004] A.E. Hassan, R. Holt. The Small World of Software Reverse Engineering. In Proc. of the 2004 Working Conference on Reverse Engineering (WCRE'04). Nov. 2004. pp. 278-283.

[Herbsleb and Grinter, 1999] J.D. Herbsleb, R.E. Grinter. Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Software, 16(5), Sept./Oct. 1999, pp. 63-70.

[Heuzeroth et al, 2002] D. Heuzeroth, T. Holl, W. Löwe. Combining Static and Dynamic Analyses to Detect Interaction Patterns. In Proc. of the Sixth International Conference on Integrated Design and Process Technology (IDPT), June 002.

[Heuzeroth et al, 2003] D. Heuzeroth, T. Holl, G. Högström, W. Löwe. Automatic Design Pattern Detection. In Proc. of the 11th International Workshop on Program Comprehension, 2003, co-located with 25th International Conference on Software Engineering, 2003.

[Hofmeister et al, 1999] C. Hofmeister, R.L. Nord, D. Soni. Describing Software Architectures with UML. In Proc. of the First Working IFIP Conference on Software Architecture, 1999.

[Holt, 1997] R. Holt. Software Bookshelf: Overview and construction. Mar. 1997, swag.uwaterloo.ca/pbs/papers/bsbuild.html

[Horowitz and Munson, 1984] Horowitz, E. and Munson, J. B. An Expansive View of Reusable Software. IEEE Transactions on Software Engineering, volume SE-10, Sept. 1984, pp. 479-487.

[IEEE 1471] Recommended Practice for Architectural Description of Software Intensive Systems. ANSI/IEEE Standard 1471-2000.

[IEEE 610] IEEE Standard Glossary of Software Engineering Terminology (R2002). IEEE Standard 610.12-1990.

[ISO 12207] R. Singh. International Standard ISO/IEC 12207: Software Life Cycle Processes. June 1998.

[Ivkovic and Godfrey, 2002] I. Ivkovic, M.W. Godfrey. Architecture Recovery of Dynamically Linked Applications: A Case Study. In Proc. of the 10th International Workshop on Program Comprehension 2002 (IWPC 2002), Jun. 2002.

[Jahnke et al, 1997] J. Jahnke, W. Schäfer, A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In Proc. of the 6th European Software Engineering Conference ESEC/FSE, 1997.

[Jain et al, 1999] A.K. Jain, M.N. Murty, P.J. Flynn. Data Clustering: A Review. In ACM Computing Surveys, 31(3), Sept. 1999, pp. 264-323.

[Java, 2005] Java Technology homepage. Mar. 2005. http://java.sun.com/

[Jonyer et al, 2001] Jonyer, I., Cook, D. J., Holder, L. B. Graph-Based Hierarchical Conceptual Clustering. In Journal of Machine Learning Research, 2, 2001, pp. 19-43.

[Kannan et al, 2004] R. Kannan, S. Vempala, A. Vetta. On Clusterings: Good, Bad and Spectral. In Journal of the ACM, 51(3), May 2004, pp. 497–515.

[Kersemakers, 2005] R. Kersemakers. Architectural Pattern Recovery. Master's thesis, Technische Universiteit Eindhoven, Jan. 2005.

[Klaus, 2002] M. Klaus. Simplifying Code Comprehension for Legacy Code Reuse. Embedded Developers Journal, Apr. 2002, pp. 8-13.

[Klein et al, 1999] T. Klein, U. Nickel, J. Niere, and A. Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, Sept. 1999.

[Korn et al, 1999] J. Korn, Y. Chen, E. Koutsofios. Chava: Reverse Engineering and Tracking of Java Applets. In Proc. of the sixth Working Conference on Reverse Engineering, Oct. 1999, pp. 314-326.

[Koschke, 2000] R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. Institut für Informatik, Universität Stuttgart, 2000.

[Koskinen, 2004] J. Koskinen. Software Maintenance Costs. University of Jyväsklä, Finland, Sept. 2004.

[Krämer and Prechtelt, 1996] C. Krämer, L. Prechtelt. Design recovery by automated search for structural design patterns in object-oriented software. In Proc. of the Working Conference on Reverse Engineering, 1996, pp. 208-215.

[Krikhaar et al, 1999] R. Krikhaar, A. Postma, A. Sellink, M. Stroucken, C. Verhoef. A Two-phase Process for Software Architecture Improvement. In Proc. of the International Conference on Software Maintenance (ICSM'99), Aug./Sept. 1999, p. 371.

[Kruchten, 1995] P. Kruchten. Architectural Blueprints – The "4+1" View Model of Software Architecture. In IEEE Software, 12 (6), Nov. 1995, pp. 42-50.

[Kuznetsov and Obëdkov, 2001] S.O. Kuznetsov, S.A. Obëdkov. Comparing performance of algorithms for generating concept lattices. In Proc. of the 9th IEEE International Conference on Conceptual Structures (ICCS '01), July 2001, pp. 35-47.

[Lange, 2003] C.F.J. Lange. Empirical Investigations in Software Architecture Completeness. Master's Thesis, Technische Universiteit Eindhoven, Department of Mathematics and Computing Science, Sept. 2003.

[Lanza, 2003a] M. Lanza. Object-Oriented Reverse Engineering. Ph.D. thesis, Universität Bern, May 2003.

[Lanza, 2003b] M. Lanza. CodeCrawler — Lessons Learned in Building a Software Visualization Tool. In Proc. of CSMR 2003, 2003, pp. 409-418.

[Lehman, 1996] M. Lehman. Laws of Software Evolution Revisited. In Proc. of the Fifth European Workshop in Software Process Technology (EWSPT'96), 1996, pp. 108-124.

[Lentz, 2004] A. Lentz. MySQL Storage Engine Architecture, Part 2: An In-Depth Look. Apr. 2004. http://dev.mysql.com/tech-resources/articles/storage-engine/part_2.html

[Lientz and Swanson, 1981] B.P. Lientz, E.B. Swanson. Software Maintenance Management. Addison- Wesley, Reading, MA, USA, 1981.

[Lindig, 2002] C. Lindig. Fast Concept Analysis. In G. Stumme, editors, Working with Conceptual Structures - Contributions to ICCS 2000, Shaker Verlag, Aachen, Germany, 2000.

[Keller et al, 1999] R.K. Keller, R. Schauer, S. Robitaille, P. Pagé. Pattern-Based Reverse-Engineering of Design Components. In Proc. of the 21st International Conference on Software Engineering (ICSE'99), May 1999, pp. 226-235.

[Kiran et al, 1997] G. Aditya Kiran, S. Haripriya, Pankaj Jalote. Effect of Object Orientation on Maintainability of Software. In Proc. of the Int. Conference on Software Maintenance, 1997, pp. 114-121.

[Klocwork, 2005] Klocwork inSight. Jan. 2005, http://www.klocwork.com/products/insight.asp

[Lakhotia, 1996] A. Lakhotia. A unified framework for expressing software subsystem classification techniques. In Journal of Systems and Software, 36, Mar. 1997, pp. 211-231.

[Macl and Havanas, 1990] D. Mancl, W. Havanas, A Study of the Impact of C++ on Software Maintenance, In Proc. of the IEEE Conference on Software Maintenance, Nov. 1990, pp. 63-69.

[Mancoridis et al, 1999] S. Mancoridis, B.S. Mitchell, Y. Chen, E.R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In Proc. IEEE International Conference on Software Maintenance (ICSM'99), 1999, pp. 50-59.

[Mansurov and Campara, 2003] N. Mansurov, D. Campara. Extracting high-level architecture from existing code with summary models. Applied Informatics 2003, pp.905-912.

[McKee, 1984] J.R. McKee. Maintenance as a function of design. In Proc. of AFIPS National Computer Conference, 1984, pp. 187-193.

[Mens and Tourwé, 2004] K. Mens and T. Tourwé. Conceptual Code Mining. To appear in Computer Languages, Systems & Structures. Submitted in May 2004.

[Mendonça, 1999] N. C. Mendonça. Software Architecture Recovery for Distributed Systems. PhD Thesis, Imperial College, Department of Computing, London, Nov. 1999.

[Meyer, 1998] B. Meyer. Object-Oriented Software Construction, 2nd edition. Prentice-Hall, NJ, 1998.

[Michaud et al, 2001] J. Michaud, M.A. Storey, H. Müller. Integrating Information Sources for Visualizing Java Programs. In Proc. of the International Conference on Software Maintenance (ICSM'01), 2001, p. 250.

[Mitchell, 2002] B.S. Mitchell. A Heuristic Search Approach to Solving the Software Clustering Problem. PhD thesis, Drexel University, Mar. 2002.

[Mitchell and Mancoridis, 2001] B.S. Mitchell, S. Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms Using Similarity Measurements. In Proc. of the International Conference on Software Maintenance (ICSM'01), Nov. 2001.

[Mitchell and Mancoridis, 2002] B.S. Mitchell, S. Mancoridis. Using Heuristic Search Techniques to Extract Design Abstractions from Source Code. In Proc. of the Genetic and Evolutionary Computation Conference (GECCO'02), Jul. 2002.

[Moad, 1990] J. Moad. Maintaining the competitive edge. Datamation 61, 62, 64, 66, 1990.

[MSVC, 2005] Microsoft Visual C++ Developer Center. Apr. 2005, http://msdn.microsoft.com/visualc/

[Müller et al, 1993] H. Müller, M. Orgun, S. Tilley, J. Uhl, A Reverse Engineering Approach To Subsystem Structure Identification. In Journal of Software Maintenance: Research and Practice, 5, 1993, pp. 181-204.

[Müller and Klashinsky, 1988] H.A. Müller, K. Klashinsky. Rigi - A System for programming-in-the-large. In Proc. of the 10th International Conference on Software Engineering (ICSE'88), 1988, pp. 80-86.

[Müller and Uhl, 1990] H.A. Müller, J.S. Uhl. Composing Subsystem Structures using (k,2)-partite Graphs. In Proc. of the Conference on Software Maintenance, Nov 1990, pp. 12-19.

[MySQL, 2005a] MySQL: The world's most popular open source database. Mar. 2005. http://www.mysql.com/

[MySQL, 2005b] MySQL Connector/J. Mar. 2005, http://www.mysql.com/products/connector/j/

[Naur and Randell, 1968] P. Naur, B. Randell Editors, Software Engineering: Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, Belgium, 1968.

[Nelson, 1996] M.L. Nelson. A Survey of Reverse Engineering and Program Comprehension. In ODU CS 551 – Software Engineering Survey, 1996, p. 2.

[Niere et al, 2001] J. Niere, J.P. Wadsack, L. Wendehals. Design Pattern Recovery Based on Source Code Analysis with Fuzzy Logic. Technical Report tr-ri-01-222, University of Paderborn, 2001.

[Niere et al, 2003] J. Niere, J.P. Wadsack, L. Wendehals. Handling Large Search Space in Pattern based Reverse Engineering. In Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC'03), 2003, p. 274.

[North and Koutsofios, 1994] S. North, E. Koutsofios. Applications of Graph Visualization. In Proc. Graphics interface, 1994, pp. 235-245.

[Nosek and Palvia, 1980] J. T. Nosek, P. Palvia. Software Maintenance Management: Changes in the Last Decade. In Journal of Software Maintenance, 2(3), Sept. 1990, p. 157-174.

[O'Brien et at, 2002] L. O'Brien, C. Stoermer, C. Verhoef. Software Architecture Reconstruction: Practice Needs and Current Approaches. SEI Technical Report CMU/SEI-2002-TR-024, Software Engineering Institute, Carnegie Mellon University, Aug. 2002.

[Paakki et al, 2000] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, A.I. Verkamo. Software Metrics by Architectural Pattern Mining. In Proc. of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress), Beijing, China, Aug. 2000, pp. 325-332.

[Pal and Mitra, 2004] S.K. Pal, P. Mitra. Patterns Recognition Algorithms for Data Mining. Chapman & Hall/CRC, 2004.

[Parnas, 1972] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. In Communications of the ACM, 15(12), Dec. 1972, pp. 1053-1058.

[Parnas et al, 1984] D.L. Parnas, P.C. Clements, D.M. Weiss. The Modular Structure of Complex Systems. In Proc. of the 7th International Conference on Software Engineering (ICSE'84), Mar. 1984, pp. 408-417.

[PBS, 2005] Portable Bookshelf. Feb. 2005, http://swag.uwaterloo.ca/pbs.

[Prechtelt et al, 2001] L. Prechtelt, B. Unger, W.F. Tichy, P. Brössler, L.G. Votta. A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. In IEEE Transactions on Software Engineering, 27(12), Dec. 2001, p. 1134-1144.

[Prechtelt et al, 2002] L. Prechtelt, B. Unger-Lamprecht, M. Philippsen, W.F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. In IEEE Transactions on Software Engineering, June 2002, pp. 595-606.

[Pressman, 1992] R.S. Pressman. Software Engineering: A Practitioner's Approach. Third edition. McGraw-Hill International Editions, 1992.

[Quilici, 1995] A. Quilici. Reverse Engineering of Legacy Systems: A Path Towards Success. In Proc. of the 17th International Conference on Software Engineering (ICSE'95), Apr. 1995, pp. 333-336.

[Rigi, 2004] Rigi: A visual tool for understanding legacy systems. Dec. 2004, http://www.rigi.csc.uvic.ca/.

[Riva, 2000] C. Riva. Reverse Architecting: An Industrial Experience Report. In Proc. of the Seventh Working Conference on Reverse Engineering (WCRE'00), Nov. 2000, pp. 42-50.

[Rumbaugh, 1990] J.R. Rumbaugh, M.R. Blaha, W. Lorensen, F. Eddy, W. Premerlani. Object-Oriented Modeling and Design. Prentice Hall, 1st edition Oct. 1990.

[Sartipi, 2001] K. Sartipi. Alborz: A query-based tool for software architecture recovery. In Proc. of the IEEE International Workshop on Program Comprehension, May 2001, pp. 115-116.

[Sartipi and Kontogiannis, 2002] K. Sartipi, K. Kontogiannis. A user-assisted approach to component clustering. In Journal of Software Maintenance: Research and Practice (JSM), Jul./Aug. 2003, vol. 15, issue 4, pages 265-295.

[Sartipi and Kontogiannis, 2003] K. Sartipi, K. Kontogiannis. Pattern-based Software Architecture Recovery. In Proc. of the Second ASERC Workshop on Software Architecture, Feb. 2003.

[Schwanke, 1991] R.W. Schwanke. An Intelligent Tool for Re-engineering Software Modularity. In Proc. of the 13th International Conference on Software Engineering, 1991, pp. 83-92.

[SEI, 2003] Software Engineering Institute. How Do You Define Software Architecture? Jan. 2005, http://www.sei.cmu.edu/architecture/definitions.html

[Shokoufandeh et al, 2004] A. Shokoufandeh, S. Mancoridis, T. Denton, M. Maycock. Spectral and meta-heuristic algorithms for software clustering. In Journal of Systems and Software, accepted Mar. 2004, published online.

[Shtern and Tzerpos, 2004] M. Shtern, V. Tzerpos. A Framework for Comparison of Nested Software Decompositions. In Proc. of the 11th Working Conference on Reverse Engineering (WCRE'04), Nov. 2004.

[Shull et al, 1996] F. Shull, W.L. Melo, V.R. Basili. An Inductive Method for Discovering Design Patterns from Object-Oriented Software Systems. Technical Report CS-TR-96-10, University of Maryland, Computer Science Department, Oct 1999.

[Siff and Reps, 1997] M. Siff, T. Reps. Identifying Modules via Concept Analysis. In Proc. of the International Conference on Software Maintenance (ICSM '97), 1997, pp. 170-179.

[Siff and Reps, 1998] M. Siff, T. Reps. Identifying Modules via Concept Analysis. In Technical Report TR-1337, Computer Sciences Department, University of Wisconsin, Madison, WI, USA, 1998.

[Silberschatz et al, 2002] A. Silberschatz, H. Korth, S. Sudarshan. Database System Concepts, 4th edition. McGraw-Hill Higher Education, 2002.

[Sim and Koschke, 2001] S. Elliott Sim, R. Koschke. WoSEF: Workshop on Standard Exchange Format. In ACM SIGSOFT Software Engineering Notes, 26, Jan. 2001, pp. 44-49.

[Snelting] G. Snelting. Concept Lattices in Software Analysis. Universität Passau.

[Snelting, 1996] G. Snelting. Reengineering of Configurations Based on Mathematical Concept Analysis. In ACM Transactions on Software Engineering and Methodology, 5(2), Apr. 1996, pp. 146-189.

[Snelting, 2000] G. Snelting. Software Reengineering Based on Concept Lattices. In Proc. of the European Conference on Software Maintenance and Reengineering (CSMR 2000), Mar. 2000, pp. 1-8.

[SNIFF+, 2005] SNiFF+. Feb. 2005, http://www.windriver.com/products/development_tools/ide/sniff_plus

[Sommerville, 2004] I. Sommerville. Software Engineering. 7th edition. Addison-Wesley, 2004.

[Storey et al, 2001] M.A. Storey, C. Best, J. Michaud. SHriMP Views: An Interactive Environment for Exploring Java Programs. In Proc. of the Ninth Int. Workshop on Program Comprehension (IWPC'01), 2001, p. 111.

[Stroustrup, 1997] B. Stroustrup. The C++ Programming Language, 3rd edition. Addison-Wesley, 1997.

[Sun, 2005] Sun Microsystems Inc. Products and Technologies: Java Technology. June 2005, http://java.sun.com/.

[Swanson and Chapin, 1995] E.B. Swanson, N. Chapin. Interview with E. Burton Swanson. In Journal of Software Maintenance 1995, 7(5), pp. 303-315.

[Tewari] R. Tewari. Empirical Investigation of Software Reuse in Object-Oriented Systems. Temple University, Philadelphia, USA.

[Tilley et al, 2003] T. Tilley, R. Cole, P. Becker , P. Eklund. A Survey of Formal Concept Analysis Support for Software Engineering Activities. In Proc. of the First International Conference on Formal Concept Analysis - ICFCA'03}, G. Stumme, Feb. 2003, Springer-Verlag.

[Tonella and Antoniol, 1999] P. Tonella, G. Antoniol. Object Oriented Design Pattern Inference. In Proc. of the International Conference on Software Maintenance (ICSM'99), Sept. 1999, pp. 230-238.

[Tonella and Antoniol, 2001] P. Tonella, G. Antoniol. Inference of Object Oriented Design Patterns. In Journal of Software Maintenance and Evolution: Research and Practice, 13(5), published online Oct., 2001, pp. 309-330.

[Trevors and Godfrey, 2002] A. Trevors, M.W. Godfrey. Architectural Reconstruction in the Dark. Position paper, Workshop on Software Architecture Reconstruction collocated with WCRE '02, Oct. 2002.

[Trifu, 2003] M. Trifu. Architecture-Aware, Adaptive Clustering of Object-Oriented Systems. Diploma thesis, Forschungszentrum Informatik, Karlsruhe, Germany, Sept. 2003.

[Turski, 1981] W. Turski. Software Stability. In Proc. of the 6th ACM Conference on Systems Architecture.

[Tzerpos, 2005] Homepage of V.B. Tzerpos, Mar. 2005. http://www.cs.yorku.ca/~bil/

[Tzerpos and Holt, 1997] V. Tzerpos, R.C. Holt. The orphan adoption problem in architecture maintenance. In Proc. of the fourth Working Conference on Reverse Engineering (WCRE'97), Oct. 1997, pp.76-84.

[Tzerpos and Holt, 1998] V. Tzerpos, R. C. Holt. Software Botryology: Automatic Clustering of Software Systems. In Proc. of the International Workshop on Large-Scale Software Composition, Aug. 1998.

[Tzerpos and Holt, 1999] V. Tzerpos, R.C. Holt. MoJo: A distance metric for software clusterings. In Proc. of the 6th Working Conference on Reverse Engineering (WCRE'99), Oct. 1999, pp. 187-193.

[Tzerpos and Holt, 2000] V. Tzerpos, R.C. Holt. ACDC: An Algorithm for Comprehension-Driver Clustering. In Proc. of the seventh Working Conference On Reverse Engineering (WCRE'00), 2000, pp. 258-267.

[Valtchev et al, 2003] P. Valtchev, D. Grosser, C. Roume, M.R. Hacene. Galicia: an open platform for lattices. In Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03), pp. 241-254, Shaker Verlag, (21-25 July) 2003.

[Viljamaa, 2002] J. Viljamaa. Automatic Extraction of Framework Specialization Patterns. Licentiate thesis, Report C-2002-47, Department of Computer Science, University of Helsinki, 2002.

[Viljamaa, 2003] J. Viljamaa. Reverse Engineering Framework Reuse Interfaces. In Foundations of software Engineering, Proc. of the 9th European Engineering Conference held jointly 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2003, pp. 217-226.

[Wen and Tzerpos, 2003] Z. Wen. V. Tzerpos. An optimal algorithm for MoJo distance. In Proc. of the 11th International Workshop n Program Comprehension (IWPC'03), May 2003, pp. 227-235.

[Wen and Tzerpos, 2004a] Z. Wen, V. Tzerpos. Evaluating similarity measures for software decompositions. In Proc. of the International Conference on Software Maintenance (ICSM'04), Sept. 2004, pp. 368-377.

[Wen and Tzerpos, 2004b] Z. Wen, V. Tzerpos. An effectiveness measure for software clustering algorithms. In Proc. of the 12th International Conference on Program Comprehension (IWPC'04), Jun. 2004, p. 194

[Wendehals, 2003] L. Wendehals. Improving Design Pattern Instance Recognition by Dynamic Analysis. In Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), May 2003.

[Wiggerts, 1997] T.A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In Proc. of the Fourth Working Conference on Reverse Engineering (WCRE '97), 1997, p. 33.

[Wille, 1981] R. Wille. Restructuring lattice theory: An approach based on hierarchies of concepts. In I. Rival, editor, Ordered Sets, pp. 445-470. NATO Advanced Study Institute, Sept. 1981.

[Wong, 1998] K. Wong. Rigi User's Manual, version 5.4.4, Jun. 1998.

[Wuyts, 1998] R. Wuyts. Declarative Reasoning about the Structure Object-Oriented Systems. In Proc. of the TOOLS USA '98 Conference, 1998, pp. 112-124.

[XDR, 2005] XML Standards Reference. Feb. 2005, msdn.microsoft.com/library/default.asp?url=/library/en-us/xmlsdk/html/xmconXDR.asp.

[XML, 2005] Extensible Markup Language (XML). Apr. 2005, http://www.w3.org/XML/

[XSLT, 2005] XSL Transformations (XSLT). W3C Recommendation 16 Nov. 1999, http://www.w3.org/TR/xslt.

[Zelkowitz et al, 1979] M. Zelkowitz, A. Shaw, J. Gannon. Principles of Software Engineering and Design. Prentice-Hall, 1979.

[Zhao, 2000] J. Zhao. A Slicing-Based Approach to Extracting Reusable Software Architectures. In Proc. of the 4th European Conference on Software Maintenance and Reengineering, Feb. 2000, pp. 215-223.

Appendix 2 Schema of fact extraction output

This appendix describes the XDR schema [XDR, 2005] of the output of the "fact extraction" module of the pattern detection prototype. The "context generation" module takes XML satisfying this schema as input.

```
<?xml-stylesheet type="text/xsl" href="xdr-schema-NoSource.xsl"?>

<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="Model" content="eltOnly" order="one">
      <element type="Classes"/>
      <element type="Relations"/>
</ElementType>

<ElementType name="Classes" content="eltOnly" order="one">
      <element type="Class"/>
</ElementType>

<ElementType name="Class" order="many">
      <attribute type="xmi.id"/>
      <attribute type="name"/>
</ElementType>

<ElementType name="Relations" content="eltOnly" order="one">
      <element type="A"/>
      <element type="I"/>
      <element type="C"/>
</ElementType>

<ElementType name="A" order="many">
      <attribute type="C1"/>
      <attribute type="C2"/>
</ElementType>

<ElementType name="I" order="many">
      <attribute type="C1"/>
      <attribute type="C2"/>
</ElementType>

<ElementType name="C" order="many">
      <attribute type="C1"/>
      <attribute type="C2"/>
</ElementType>

<AttributeType name="C1" dt:type="string"/>

<AttributeType name="C2" dt:type="string"/>

<AttributeType name="xmi.id" dt:type="string"/>

<AttributeType name="name" dt:type="string"/>

</Schema>
```

Appendix 3 Galicia import schema

This appendix describes the import format of Galicia for binary contexts. The “context generation” module of the pattern detection prototype produces XML that satisfies this XDR schema [XDR, 2005].

```
<?xml-stylesheet type="text/xsl" href="xdr-schema-NoSource.xsl"?>

<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="BIN" content="eltOnly" order="one">
      <element type="OBJS"/>
      <element type="ATTS"/>
      <element type="RELS"/>
</ElementType>

<ElementType name="OBJS" content="eltOnly" order="one">
      <element type="OBJ"/>
</ElementType>

<ElementType name="OBJ" order="many" content="textOnly">
      <attribute type="id"/>
</ElementType>

<ElementType name="ATTS" content="eltOnly" order="one">
      <element type="ATT"/>
</ElementType>

<ElementType name="ATT" order="many" content="textOnly">
      <attribute type="id"/>
</ElementType>

<ElementType name="RELS" content="eltOnly" order="one">
      <element type="REL"/>
</ElementType>

<ElementType name="REL" order="many">
      <attribute type="idObj"/>
      <attribute type="idAtt"/>
</ElementType>

<AttributeType name="id" dt:type="number"/>

<AttributeType name="idObj" dt:type="number"/>

<AttributeType name="idAtt" dt:type="number"/>

</Schema>
```

Appendix 4 Galicia export schema

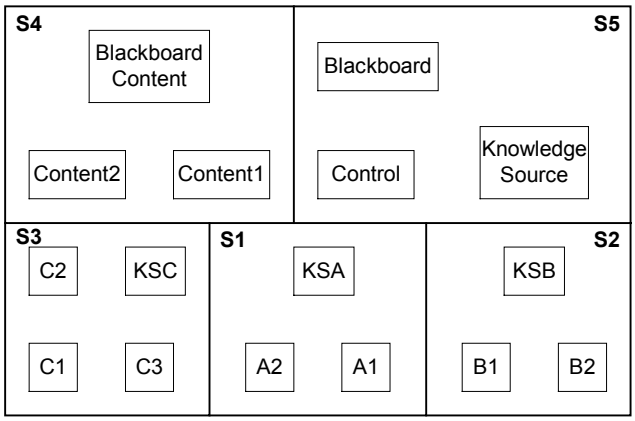
This appendix describes the XML export format Galicia uses for concept lattices with an XDR schema [XDR, 2005].

```
<?xml-stylesheet type="text/xsl" href="xdr-schema-NoSource.xsl"?>
<Schema xmlns="urn:schemas-microsoft-com:xml-data"
xmlns:dt="urn:schemas-microsoft-com:datatypes">
<ElementType name="LAT" content="eltOnly" order="one" model="closed">
      <attribute type="Desc"/>
      <attribute type="type"/>
      <element type="MINSUPP"/>
      <element type="OBJS"/>
      <element type="ATTS"/>
      <element type="NODS"/>
</ElementType>
<ElementType name="MINSUPP" order="one" content="textOnly">
      <attribute type="id"/>
</ElementType>
<ElementType name="OBJS" content="eltOnly" order="one">
      <element type="OBJ"/>
</ElementType>
<ElementType name="OBJ" order="many" content="textOnly">
      <attribute type="id"/>
</ElementType>
<ElementType name="ATTS" content="eltOnly" order="one">
      <element type="ATT"/>
</ElementType>
<ElementType name="ATT" order="many" content="textOnly">
      <attribute type="id"/>
</ElementType>
<ElementType name="NODS" content="eltOnly" order="one">
      <element type="NOD"/>
</ElementType>
<ElementType name="NOD" order="many">
      <attribute type="id"/>
      <element type="EXT"/>
      <element type="INT"/>
      <element type="SUP_NOD"/>
</ElementType>
<ElementType name="EXT" content="eltOnly" order="one">
      <element type="OBJ"/>
</ElementType>
<ElementType name="INT" content="eltOnly" order="one">
      <element type="ATT"/>
</ElementType>
<ElementType name="SUP_NOD" content="eltOnly" order="one">
      <element type="PARENT"/>
</ElementType>
<ElementType name="PARENT" content="eltOnly" order="one">
      <attribute type="id"/>
</ElementType>
<AttributeType name="id" dt:type="number"/>
<AttributeType name="Desc" dt:type="string"/>
<AttributeType name="type" dt:type="strng"/>
</Schema>
```

Appendix 5 Architect decompositions

This appendix describes the decompositions produced by ten architects and designers of the simple-blackboard application, which is described in paragraph 7.3.2. The decompositions are shown in a simplified version of the class diagram, which places the classes in the same way as Figure 39. The boxes around groups of classes denote the subsystems. In case a hierarchical decomposition was produced, the hierarchy is visualised through containment of the boxes. Besides the actual decomposition, the considerations that led to the decomposition are also described.

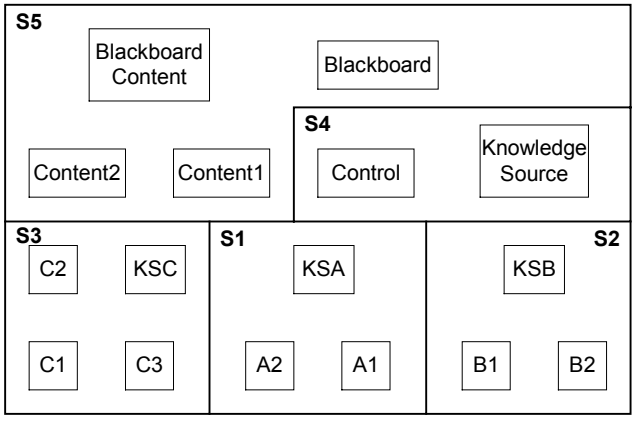
1 Project organisation and class functionality were the primary criteria. Each team represents a subsystem that is testable in isolation (drivers/stubs).



First split in blackboard and clients. Three client-subsystems are identified, S1, S2, S3. Blackboard split in data and control part, S4 and S5. Content1 and Content2 are not part of a specific client, so they were added to S4.

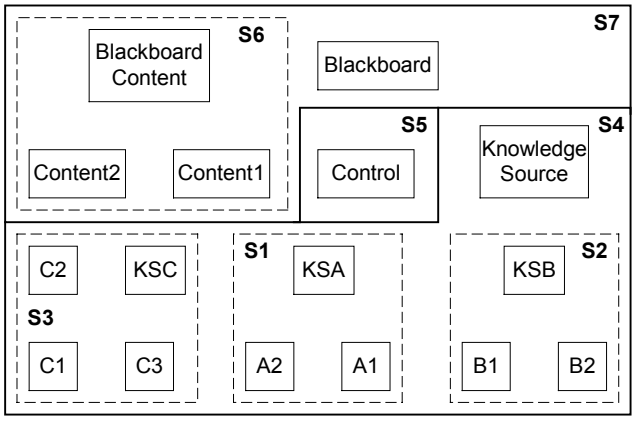
No hierarchy needed. Only generalisation and inheritance are considered; no dependencies.

2 Class functionality was the primary criterion. Three client subsystems are identified that are conceptually equivalent (S1, S2, S3). Each is an object KSx with helper objects.



Blackboard is split in data and control part (S5 and S4). Content1 and Content2 are not part of a specific client. Because they are not related they are not placed together in a subsystem. Since they are small a separate subsystem for each is not necessary. Since they are related to the data they were added to S5.

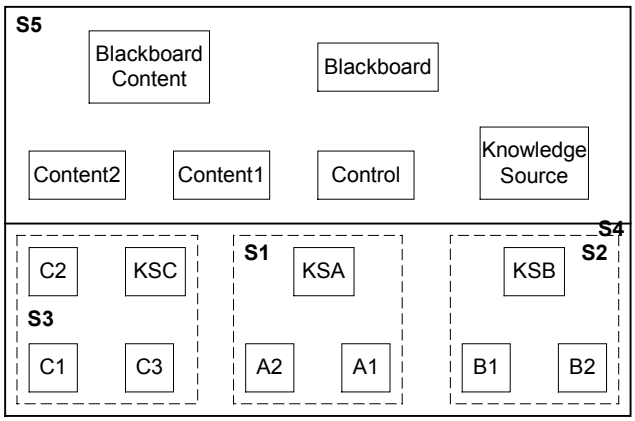
3 Inter-class relations and class functionality were the primary criteria. The priority of the relations, sorted from very important to unimportant, is: composition, inheritance, association, dependency. The last two were not considered while building the decomposition.



Composition relations led to S1 and S2. S3 implements comparable functionality. All client-related classes together form S4. S6 is formed because of the inheritance relation with BlackboardContent. For functional reasons S6 and Blackboard together form S7. S5 contains the remaining class.

4 Initially a client-server view was applied and the role of the classes was considered. The blackboard classes represent the server and the others the clients. This led to S4 and S5.

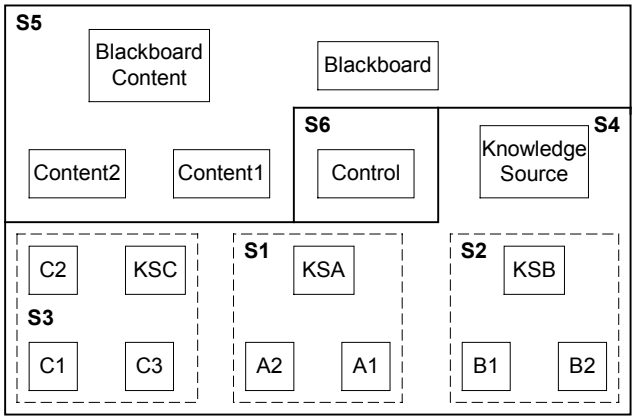
Next, all three clients are clients of the blackboard, and are not directly related to each other. Therefore S4 is decomposed further into S1, S2, S3.



5 Main decomposition criteria were functionality, exchangeability and allocation to development teams.

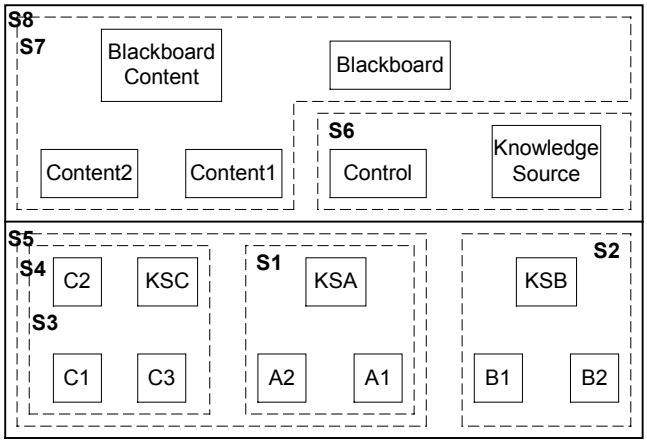
Initially a data (passive, S5), processing (active, S4) and remaining part (S6) are distinguished. Functional arguments led to S1, S2 and S3. These are exchangeable processing components.

S5 is not split further.

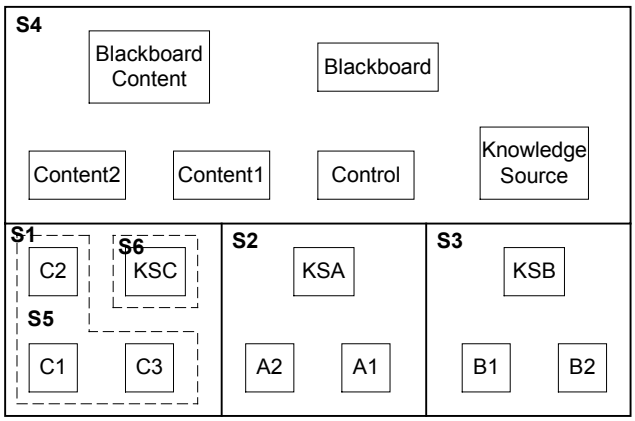


6 Class functionality was the main decomposition criterion.

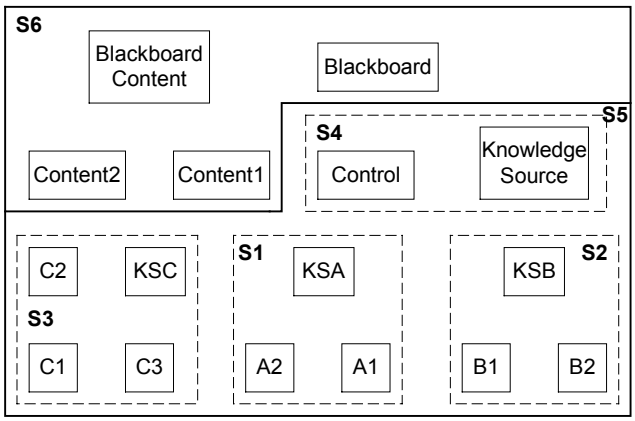
At the top level a division in generic (S8) and specific classes (S5) is made. S8 is split in a data (S7) and a processing part (S6). S5 is split in an I/O (S4) and a processing part (S2). S4 is further split in an input (S3) and an output part (S1).



7 The decomposition criteria were functional differences, generality, volatility and to a lesser extent size. Initial division in generic and specific parts. Volatile classes are placed together to isolate change. Small non-volatile parts that do not justify a separate subsystem are placed with generic parts. This led to S4 and S1, S2 and S3.
S2 is not split because it is not very complex. S1 is split into S5 and S6 because of the inheritance relationship; S5 apparently is a generic part



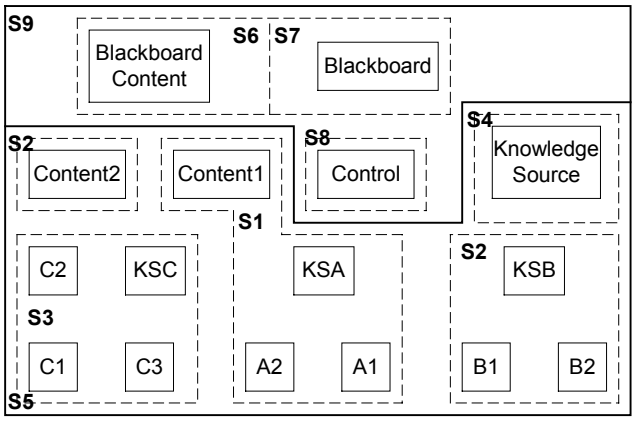
8 Development considerations and functional arguments were the main criteria. Subsystem boundaries are placed where well-defined interfaces are desired. All three relation-types are considered equally important, but execution-dependencies are more important than the other dependencies.



A client (S5) and a data part (S6) are distinguished. Clients have relationships with almost everything except each other, giving S1, S2 and S3. Content is simple, so part of S6. Clients have much interaction with KnowledgeSource, so separate subsystem (S4). Control could be part of S6 or S4. S4 is chosen.

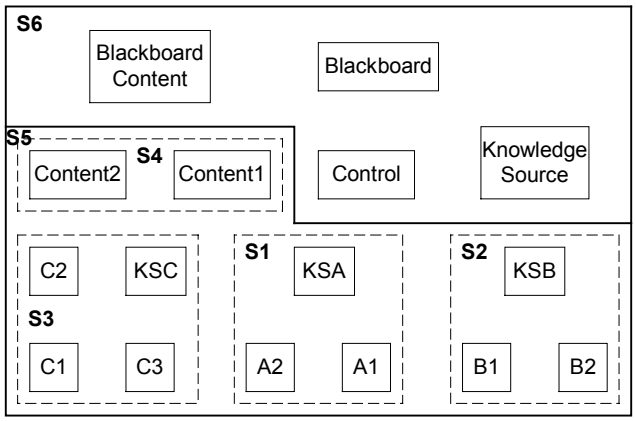
Alternative that was considered but not chosen: KSA, KSB and KSC form an interface to the data. If they are generic A1, A2, B1, B2. C1, C2 and C3 could be placed in a library.

9 Functionality was the main criterion, together with the blackboard style. Subsystem decomposition in data, control and algorithm parts (S9 and S5, no division algorithm-control found).



Three clients of data part found, S1, S2 and S3. In case of a client-server relation the server defines the interface (unless there are many servers), so Content1 and Content2 are added to their producers (S1 and S2). S4 is put separate since it contains a generic interface that is implemented by KSA, KSB and KSC. If its classes are large, S9 is split into S6, S7 and S8.

10 Initial division in a generic (S6) and a specific (S5) part with KnowledgeSource and BlackboardContent as re-use interfaces. A blackboard structure with clients is recognised.



Functional dependencies for clients led to three clients, S1, S2 and S3. All three relationship-types were taken into account.
Content1 and Content2 are not associated with a single client, so they are added to special subsystem (S4).

Appendix 6 Clustering results for Grizzly & RIP Worker

This appendix gives the fifty parameter-tuples that produced the best clustering results for Grizzly and the RIP Worker (see paragraph 7.3.2). Note that this does not necessarily include the optimal parameter-tuple because a subset of the search-space has been investigated.

The columns labelled "p_{w_a}","p_{w_g}","p_{w_d}","p_c" and "p_i" contain the five user-specified parameters of the Bunch Export module. The values of the boolean parameters, p_c, p_i, are denoted as "T" and "F" for true and false respectively. The "MQ" and "EM" columns refer to the MoJoQuality and EdgeMoJo metrics respectively. The column labelled "AV" contains the average value of the metric for ten different clusterings. The column labelled "SD" contains the standard deviation for these clusterings. The parameter-tuples are sorted ascending according to the EdgeMoJo value. The left side of the table contains the best 25 tuples, and the right side the other ones.

Grizzly

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
2	5	5	F	F	69	1,5	102	8,0
1	3	2	F	F	69	1,3	102	6,4
0	0	6	T	F	70	1,8	102	8,9
2	3	5	F	T	68	1,5	102	8,5
1	4	3	F	F	69	1,1	103	6,0
3	6	5	F	T	68	1,4	103	8,0
4	4	3	F	F	68	1,2	103	7,6
3	2	4	F	T	69	1,0	103	5,7
6	5	4	F	T	69	0,9	103	6,7
1	2	3	F	T	68	1,5	103	7,3
3	3	1	F	F	68	1,3	103	6,3
6	2	5	F	T	69	1,6	103	7,3
3	6	1	F	F	69	1,0	103	3,7
0	0	2	F	F	70	1,2	104	5,1
6	2	1	T	T	69	0,9	104	6,3
1	5	5	T	T	69	1,2	104	4,1
4	2	5	F	F	69	0,9	104	5,8
6	1	6	F	T	68	0,8	104	5,6
0	0	1	T	F	70	1,3	104	5,4
2	1	2	T	F	68	0,9	104	5,9
1	3	6	T	F	69	1,5	104	4,7
2	1	2	F	F	68	1,3	104	6,7
1	5	3	T	F	69	1,3	104	6,9
2	6	3	T	F	68	1,6	104	6,7
4	4	4	T	T	68	1,1	104	5,0

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ ()		EM	
					AV	SD	AV	SD
1	1	6	T	F	69	1,2	104	6,7
3	1	4	T	T	68	1,6	104	8,4
3	1	6	T	T	68	0,9	104	3,8
1	6	6	F	F	68	0,9	104	4,8
4	4	1	T	F	68	1,7	104	8,7
6	2	6	F	T	69	1,2	104	6,4
4	1	4	T	F	69	1,7	104	6,1
4	5	2	T	T	68	1,4	104	6,7
5	6	2	F	F	68	1,5	104	6,6
5	5	5	T	F	69	1,1	104	4,2
5	6	2	T	T	68	1,6	104	7,5
5	5	4	T	T	68	1,5	104	7,4
6	4	3	F	T	69	1,3	104	4,5
1	1	3	T	F	69	1,5	105	6,8
6	4	1	T	T	68	1,5	105	6,3
6	3	6	T	F	68	1,5	105	7,5
4	6	6	T	T	69	1,0	105	3,8
1	6	6	T	F	68	1,1	105	8,6
6	1	2	T	T	68	1,2	105	3,0
1	4	5	T	T	69	1,7	105	5,3
5	2	5	T	F	69	1,3	105	4,5
3	4	4	T	T	68	1,6	105	6,3
2	1	5	T	F	69	1,1	105	3,1
2	6	5	F	F	69	0,9	105	3,1
1	1	6	F	F	68	1,5	105	7,7

Table 30: The fifty parameter-tuples that give the best clustering of Grizzly

RIP Worker

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
0	5	2	F	F	66	2,1	43	1,6
2	4	3	F	T	66	2,2	43	1,7
1	6	4	T	T	67	2,6	43	2,0
1	5	2	F	T	67	2,0	43	1,3
2	4	1	T	F	66	2,9	43	1,8
1	4	1	T	T	66	2,8	43	1,7
0	1	1	T	F	66	2,3	43	2,3
1	5	6	F	T	65	1,9	43	1,0
6	3	4	T	F	66	2,0	43	1,4
1	1	5	T	F	66	2,0	43	1,7
3	3	2	T	T	66	2,3	43	2,2
3	3	4	T	T	66	2,0	43	1,1
4	1	1	T	F	66	2,5	43	1,9
4	4	4	T	F	67	1,6	43	1,5
5	1	6	T	T	65	2,4	43	1,4
6	1	6	F	F	65	2,2	43	1,5
4	6	1	T	T	66	2,0	43	1,3
4	5	4	T	F	67	1,9	43	1,9
3	1	2	F	T	66	3,0	43	2,6
4	6	2	F	T	65	2,3	43	1,2
1	4	1	T	F	66	2,6	43	1,7
0	1	3	T	F	67	2,6	43	2,7
4	4	5	F	F	67	2,4	43	3,0
2	1	2	F	F	65	1,7	43	1,7
1	1	6	T	F	66	1,8	43	1,2

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ ()		EM	
					AV	SD	AV	SD
5	5	4	T	T	66	1,7	43	1,6
3	5	1	F	F	66	2,4	43	2,1
0	4	1	F	F	66	2,1	43	1,6
1	1	6	F	T	66	2,3	43	1,9
5	2	3	T	T	65	2,7	43	1,9
3	1	1	T	F	66	2,7	43	2,0
3	2	4	T	T	65	1,9	43	2,5
5	6	5	F	F	65	1,8	43	1,2
5	3	3	F	T	67	2,3	43	1,8
5	3	4	T	F	66	1,7	43	1,2
2	5	2	T	T	65	2,3	43	1,5
0	2	6	T	F	65	2,1	43	1,3
2	4	2	T	T	65	2,6	43	1,7
0	6	4	F	F	66	1,9	43	2,6
2	4	4	F	F	66	2,4	43	2,4
3	2	6	T	F	66	3,1	43	3,7
4	4	2	T	T	66	3,1	43	4,2
1	5	4	F	F	66	2,4	43	2,1
3	3	3	T	T	65	2,2	43	1,9
2	3	3	T	T	66	2,6	43	2,3
3	6	6	F	F	66	1,9	43	2,0
4	3	4	F	T	66	2,3	43	2,4
0	2	4	T	F	65	2,4	43	2,0
2	5	1	F	T	65	1,8	43	1,2
0	2	5	F	F	65	2,1	44	1,6

Table 31: The fifty parameter-tuples that give the best clustering of the RIP Worker

Appendix 7 Clustering results for Océ Controller

This appendix gives the clustering results for the Océ Controller (see paragraph 7.4), using the same notation as in Appendix 6. Note that this does not necessarily include the optimal parameter-tuple because a subset of the search-space has been investigated.

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
4	6	1	T	T	61	0,6	1.639	18
1	4	1	T	T	61	0,5	1.645	17
1	5	5	T	T	60	0,6	1.646	24
1	1	5	T	F	60	0,5	1.647	23
2	5	5	F	F	60	0,5	1.648	17
1	5	6	F	T	60	0,5	1.649	18
3	3	4	T	T	60	0,4	1.649	23
5	1	6	T	T	60	0,3	1.650	8
3	6	5	F	T	60	0,6	1.650	17
6	5	4	F	T	60	0,4	1.652	13
6	1	6	F	T	60	0,6	1.652	15
3	2	4	F	T	60	0,5	1.652	18
6	2	5	F	T	60	0,3	1.653	10
2	4	3	F	T	60	0,6	1.653	16
1	3	2	F	F	60	0,5	1.653	18
6	3	4	T	F	60	0,5	1.653	12
1	2	3	F	T	60	0,4	1.654	8
2	4	1	T	F	60	0,5	1.654	19
4	5	4	T	F	60	0,5	1.654	16
6	1	6	F	F	60	0,3	1.655	13

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
4	6	2	F	T	60	0,4	1.655	10
3	3	2	T	T	60	0,2	1.655	7
4	1	1	T	F	60	0,4	1.656	11
3	1	2	F	T	60	0,5	1.656	20
4	4	4	T	F	60	0,5	1.656	13
4	4	3	F	F	60	0,4	1.657	12
6	2	1	T	T	60	0,4	1.657	13
2	1	2	T	F	60	0,3	1.657	19
2	3	5	F	T	60	0,5	1.658	13
1	6	4	T	T	60	0,5	1.659	16
1	4	3	F	F	60	0,4	1.659	10
1	5	2	F	T	60	0,4	1.660	14
4	2	5	F	F	60	0,9	1.670	51
3	6	1	F	F	60	1,4	1.670	63
3	3	1	F	F	60	1,2	1.671	61
0	5	2	F	F	59	2,3	1.721	97
0	1	1	T	F	59	1,6	1.730	79
0	0	6	T	F	58	0,3	1.773	15
0	0	1	T	F	58	0,4	1.775	21
0	0	2	F	F	58	0,4	1.779	15

Table 32: Clustering result for version 7e of the Océ Controller

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
4	6	1	T	T	62	0,5	1.477	30
2	1	2	T	F	62	0,5	1.481	17
6	3	4	T	F	62	0,5	1.481	18
1	4	3	F	F	62	0,5	1.484	17
2	3	5	F	T	62	0,4	1.484	13
1	5	6	F	T	62	0,5	1.484	17
3	6	1	F	F	62	0,5	1.484	13
3	3	2	T	T	62	0,6	1.486	19
4	2	5	F	F	62	0,7	1.486	17
6	2	5	F	T	62	0,3	1.486	8
4	4	4	T	F	62	0,6	1.486	17
4	6	2	F	T	62	0,4	1.489	18
1	5	5	T	T	62	0,4	1.489	21
5	1	6	T	T	62	0,3	1.490	8
1	3	2	F	F	62	0,3	1.490	12
3	2	4	F	T	62	0,4	1.491	15
2	5	5	F	F	62	0,5	1.492	17
1	2	3	F	T	62	0,4	1.493	16
6	2	1	T	T	62	0,4	1.493	14
6	1	6	F	F	62	0,5	1.493	11

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
3	3	1	F	F	62	0,3	1.495	11
6	5	4	F	T	62	1,7	1.500	84
3	6	5	F	T	62	1,3	1.501	62
4	4	1	T	F	62	0,9	1.502	57
1	1	5	T	F	62	1,4	1.503	73
3	1	2	F	T	62	1,1	1.504	66
1	6	4	T	T	62	1,0	1.506	67
4	5	4	T	F	62	1,6	1.513	89
2	4	1	T	F	62	1,6	1.513	75
3	3	4	T	T	62	1,2	1.514	73
6	1	6	F	T	62	1,6	1.517	86
4	4	3	F	F	62	1,5	1.518	90
1	4	1	T	T	62	1,4	1.523	78
1	5	2	F	T	61	2,2	1.530	125
0	5	2	F	F	61	0,7	1.536	21
2	4	3	F	T	61	1,8	1.540	97
0	1	1	T	F	60	2,2	1.594	111
0	0	2	F	F	60	0,3	1.631	14
0	0	6	T	F	60	0,4	1.637	11
0	0	1	T	F	59	1,5	1.661	83

Table 33: Clustering result for version 8a of the Océ Controller

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
0	0	2	F	F	74	1,1	1.223	107
0	0	1	T	F	74	1,0	1.229	107
0	0	6	T	F	73	0,5	1.266	59
1	5	6	F	T	72	2,1	1.287	152
1	5	5	T	T	72	1,3	1.293	107
0	1	1	T	F	72	0,8	1.302	74
2	4	3	F	T	72	1,2	1.311	99
1	4	1	T	T	72	0,9	1.311	72
2	3	5	F	T	72	1,4	1.315	121

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
2	4	1	T	F	71	1,3	1.357	108
6	1	6	F	F	71	1,2	1.358	95
6	1	6	F	T	71	0,4	1.359	44
3	3	4	T	T	71	0,7	1.361	72
3	3	1	F	F	71	1,5	1.365	120
6	2	1	T	T	71	1,0	1.366	86
3	2	4	F	T	71	0,8	1.367	72
4	4	4	T	F	71	0,4	1.372	44
1	3	2	F	F	71	0,5	1.372	49

1	6	4	T	T	72	1,4	1.315	119
1	2	3	F	T	72	1,1	1.316	96
0	5	2	F	F	72	1,2	1.335	111
6	5	4	F	T	72	1,2	1.335	111
4	4	3	F	F	72	1,4	1.340	114
4	6	2	F	T	71	0,7	1.345	65
3	6	5	F	T	71	0,7	1.346	64
2	1	2	T	F	71	1,0	1.347	86
2	5	5	F	F	71	0,4	1.347	43
3	3	2	T	T	71	0,9	1.353	85
6	2	5	F	T	71	0,5	1.354	46
4	6	1	T	T	71	0,5	1.373	49
6	3	4	T	F	71	0,4	1.374	46
4	1	1	T	F	71	0,7	1.376	70
3	6	1	F	F	71	0,8	1.377	67
1	4	3	F	F	71	0,5	1.379	48
5	1	6	T	T	71	0,4	1.380	42
3	1	2	F	T	71	0,8	1.380	71
1	1	5	T	F	71	0,4	1.381	42
1	5	2	F	T	71	0,6	1.383	55
4	2	5	F	F	71	0,3	1.393	40
4	5	4	T	F	71	0,9	1.398	85

Table 34: Clustering result for class-relations-intersection of version 7e and 1

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
0	0	6	T	F	78	0,7	950	75
0	0	1	T	F	78	0,5	958	46
0	0	2	F	F	78	0,5	981	64
0	1	1	T	F	77	1,0	1.006	87
1	6	4	T	T	77	0,7	1.007	70
2	4	3	F	T	77	0,9	1.016	84
4	6	1	T	T	77	1,0	1.022	97
3	2	4	F	T	76	1,2	1.031	104
1	5	5	T	T	76	1,0	1.033	91
3	1	2	F	T	76	1,2	1.038	107
0	5	2	F	F	76	0,9	1.039	83
3	3	2	T	T	76	0,7	1.039	63
6	2	1	T	T	76	1,0	1.040	89
6	5	4	F	T	76	0,8	1.041	85
3	3	4	T	T	76	0,7	1.043	66
1	5	2	F	T	76	0,9	1.046	84
4	6	2	F	T	76	0,9	1.046	78
6	2	5	F	T	76	0,8	1.051	65
4	5	4	T	F	76	0,8	1.059	65
2	5	5	F	F	76	0,7	1.062	68

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
6	1	6	F	T	76	0,8	1.072	73
5	1	6	T	T	76	0,4	1.075	40
6	1	6	F	F	76	0,7	1.075	57
4	2	5	F	F	76	0,6	1.076	48
3	3	1	F	F	76	0,7	1.077	59
2	1	2	T	F	76	0,8	1.077	65
2	3	5	F	T	76	0,7	1.079	57
2	4	1	T	F	76	0,9	1.079	80
1	2	3	F	T	76	0,7	1.080	64
6	3	4	T	F	76	0,6	1.083	54
1	4	3	F	F	76	0,7	1.085	61
4	4	3	F	F	76	0,5	1.085	40
1	1	5	T	F	76	0,5	1.086	45
3	6	1	F	F	76	0,7	1.091	58
1	5	6	F	T	76	0,5	1.093	43
4	1	1	T	F	76	0,6	1.102	51
3	6	5	F	T	76	0,2	1.105	29
4	4	4	T	F	76	0,3	1.115	27
1	4	1	T	T	75	0,2	1.122	21
1	3	2	F	F	75	0,4	1.126	31

Table 35: Clustering result for class-relations-intersection of version 8a and 1

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
6	2	1	T	T	60	0,7	1.643	28
1	5	5	T	T	60	0,3	1.645	23
3	3	4	T	T	60	0,6	1.647	30
1	5	2	F	T	60	0,5	1.651	17
4	4	3	F	F	60	0,5	1.652	23
1	1	5	T	F	60	0,5	1.652	25
4	5	4	T	F	60	0,5	1.654	16
4	6	2	F	T	60	0,6	1.654	14
2	5	5	F	F	60	0,5	1.655	24
3	1	2	F	T	60	0,5	1.655	14
3	6	5	F	T	60	0,3	1.655	9
1	4	1	T	T	60	0,4	1.656	17
1	6	4	T	T	60	0,4	1.656	14
4	6	1	T	T	60	0,3	1.656	14
5	1	6	T	T	60	0,4	1.657	23
6	5	4	F	T	60	0,4	1.657	8
4	1	1	T	F	60	0,5	1.657	18
1	2	3	F	T	60	0,5	1.658	15
6	1	6	F	F	60	0,4	1.659	14
4	4	4	T	F	60	0,4	1.659	10

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
3	3	1	F	F	60	0,4	1.659	16
1	5	6	F	T	60	0,4	1.660	12
3	6	1	F	F	60	0,3	1.661	19
6	2	5	F	T	60	0,5	1.661	12
6	1	6	F	T	60	0,4	1.662	13
2	3	5	F	T	60	0,5	1.663	13
6	3	4	T	F	60	0,6	1.663	21
2	1	2	T	F	60	0,4	1.663	9
3	3	2	T	T	60	0,4	1.664	18
1	4	3	F	F	60	0,6	1.665	17
1	3	2	F	F	60	0,5	1.667	14
3	2	4	F	T	60	1,5	1.672	65
2	4	3	F	T	60	1,0	1.684	81
2	4	1	T	F	60	1,0	1.684	57
4	2	5	F	F	59	1,3	1.686	77
0	5	2	F	F	59	0,6	1.707	17
0	1	1	T	F	59	0,6	1.707	17
0	0	2	F	F	58	0,4	1.787	27
0	0	6	T	F	58	0,3	1.788	18
0	0	1	T	F	57	1,0	1.804	51

Table 36: Clustering result for class-relations-intersection of version 7e and 7d

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
6	1	6	F	T	59	0,3	1.643	26
4	6	2	F	T	59	0,4	1.645	27
2	4	3	F	T	59	0,4	1.649	10

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
6	2	1	T	T	59	0,4	1.662	22
6	1	6	F	F	59	0,4	1.663	15
4	4	3	F	F	59	0,4	1.663	10

6	5	4	F	T	59	0,4	1.651	12
3	3	1	F	F	59	0,3	1.652	16
1	6	4	T	T	59	0,4	1.653	17
1	2	3	F	T	59	0,3	1.653	14
1	5	5	T	T	59	0,4	1.654	16
6	2	5	F	T	59	0,5	1.655	18
3	6	1	F	F	59	0,5	1.655	12
2	5	5	F	F	59	0,4	1.657	15
1	1	5	T	F	59	0,6	1.657	13
4	1	1	T	F	59	0,4	1.657	14
2	1	2	T	F	59	0,2	1.657	13
2	4	1	T	F	59	0,4	1.658	11
4	2	5	F	F	59	0,4	1.658	13
1	5	2	F	T	59	0,5	1.659	20
3	3	4	T	T	59	0,5	1.659	13
3	2	4	F	T	59	0,3	1.659	11
3	6	5	F	T	59	0,5	1.660	14
1	4	1	T	T	59	0,3	1.664	16
1	3	2	F	F	58	0,6	1.665	17
1	5	6	F	T	59	0,5	1.666	16
4	4	4	T	F	59	0,5	1.666	19
3	1	2	F	T	59	0,5	1.666	19
4	6	1	T	T	59	0,3	1.669	15
6	3	4	T	F	58	1,3	1.669	55
4	5	4	T	F	58	1,4	1.674	68
3	3	2	T	T	58	1,7	1.675	69
5	1	6	T	T	58	1,3	1.679	54
1	4	3	F	F	58	1,3	1.682	64
2	3	5	F	T	58	1,4	1.689	73
0	1	1	T	F	58	0,6	1.704	18
0	5	2	F	F	57	0,6	1.718	18
0	0	2	F	F	56	0,6	1.799	28
0	0	1	T	F	56	0,4	1.801	26
0	0	6	T	F	56	0,5	1.812	18

Table 37: Clustering result for class-relations-intersection of version 8a and 7e

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
3	3	4	T	T	63	0,6	1.459	19
4	4	3	F	F	63	0,4	1.460	14
3	6	1	F	F	63	0,3	1.461	9
1	4	3	F	F	63	0,3	1.462	12
6	5	4	F	T	63	0,4	1.462	9
6	1	6	F	F	63	0,4	1.462	10
1	5	5	T	T	63	0,5	1.463	18
4	2	5	F	F	63	0,7	1.463	18
2	4	1	T	F	63	0,4	1.463	13
2	5	5	F	F	63	0,4	1.464	10
4	5	4	T	F	63	0,3	1.464	11
1	4	1	T	T	63	0,4	1.465	10
5	1	6	T	T	63	0,4	1.465	13
6	2	5	F	T	63	0,4	1.465	14
3	1	2	F	T	63	0,5	1.466	11
4	1	1	T	F	63	0,5	1.468	14
2	4	3	F	T	62	0,5	1.469	16
6	2	1	T	T	63	0,5	1.469	17
3	2	4	F	T	63	0,4	1.470	12
1	1	5	T	F	63	0,5	1.470	20
4	6	2	F	T	63	0,3	1.471	10
1	6	4	T	T	63	0,4	1.471	11
2	3	5	F	T	62	0,5	1.472	17
2	1	2	T	F	63	0,4	1.473	9
6	3	4	T	F	62	0,6	1.476	23
4	4	4	T	F	62	0,5	1.478	16
3	6	5	F	T	63	1,0	1.480	47
1	2	3	F	T	63	1,6	1.483	96
3	3	1	F	F	62	1,2	1.488	76
1	5	2	F	T	62	1,1	1.488	60
1	3	2	F	F	62	1,1	1.492	69
3	3	2	T	T	62	1,2	1.500	65
6	1	6	F	T	62	1,3	1.501	93
1	5	6	F	T	62	1,0	1.501	77
0	1	1	T	F	62	0,5	1.513	21
4	6	1	T	T	62	1,7	1.524	114
0	5	2	F	F	61	1,5	1.535	85
0	0	1	T	F	60	1,3	1.605	79
0	0	6	T	F	60	1,4	1.613	73
0	0	2	F	F	60	1,3	1.620	74

Table 38: Clustering result for class-relations-union of version 8a and 1

p_{w_a}	p_{w_g}	p_{w_d}	p_c	p_i	MQ (%)		EM	
					AV	SD	AV	SD
4	4	3	F	F	63	0,4	1.459	25
6	3	4	T	F	63	0,4	1.467	15
4	6	1	T	T	63	0,4	1.468	15
6	1	6	F	F	62	0,2	1.469	21
3	6	1	F	F	63	0,3	1.469	8
6	5	4	F	T	63	0,4	1.471	16
2	1	2	T	F	63	0,4	1.471	30
6	2	5	F	T	62	0,3	1.472	14
2	5	5	F	F	63	0,5	1.472	15
3	3	1	F	F	63	0,4	1.473	13
1	4	3	F	F	63	0,5	1.473	16
3	3	4	T	T	63	0,3	1.474	9
1	5	6	F	T	62	0,4	1.474	14
1	1	5	T	F	62	0,3	1.475	15
2	4	1	T	F	62	0,3	1.475	12
2	4	3	F	T	62	0,4	1.476	13
1	2	3	F	T	63	0,4	1.476	21
1	4	1	T	T	62	0,2	1.477	10
3	6	5	F	T	62	0,4	1.477	13
4	5	4	T	F	63	0,6	1.477	15
6	2	1	T	T	63	0,5	1.478	19
6	1	6	F	T	62	0,4	1.482	13
3	3	2	T	T	62	0,4	1.483	14
1	5	5	T	T	62	0,3	1.483	9
3	1	2	F	T	62	0,3	1.484	15
5	1	6	T	T	62	0,3	1.485	9
4	6	2	F	T	62	0,5	1.486	16
1	5	2	F	T	62	0,4	1.487	12
1	3	2	F	F	62	0,7	1.487	24
1	6	4	T	T	62	1,2	1.495	64
2	3	5	F	T	62	1,4	1.497	91
4	1	1	T	F	62	1,2	1.501	72
0	1	1	T	F	62	0,5	1.502	35
3	2	4	F	T	62	1,3	1.504	86
4	4	4	T	F	62	1,7	1.505	87
4	2	5	F	F	62	1,5	1.507	87
0	5	2	F	F	61	1,9	1.558	102
0	0	6	T	F	60	0,2	1.595	19
0	0	2	F	F	60	0,3	1.600	19
0	0	1	T	F	60	1,0	1.622	68

Table 39: Clustering result for class-relations-union of version 8a and 7e