Vrije Universiteit Brussel - Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes - France
and
Universidad Nacional de La Plata - Argentina
2007

Koschke Revisited

Object-Oriented Component Detection for Software Understanding

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Toon Verwaest

Promoter: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Dr. Gabriela B. Arévalo (UNLP)

Abstract

When a software engineer has to maintain a system, he needs to understand how the system is built. In order to help engineers understand existing systems, research has been conducted around automating the process of architecture recovery. A first step consists of building a straightforward browsable model of the system. However, the conceptual level of abstraction behind the design might be higher than the available level of abstraction in the used programming paradigm. Therefore, a second step which retrieves this implicit information needs to be undertaken.

In his thesis, Rainer Koschke [Kos02] has developed and evaluated several techniques which retrieve implicit architectural information from procedural systems. These techniques resulted in the detection of atomic architectural components, comparable to the concept of *prototypes*.

More and more systems are developed using the object-oriented programming paradigm. Systems built using this paradigm embed a similar, yet more coarse-grained, type of implicit information. Here we think of a higher level of abstraction, comparable to the concept of *software components*.

In this thesis, we investigate if and how some of the component detection heuristics, presented in the thesis by Koschke, can be adapted as such that they are applicable to object-oriented code in order to detect components comparable to *software components*. Additionaly, we investigate how we can complement them with available object-oriented information.

Acknowledgements

I would like to thank all the people who supported me during the research and the writing phase of this dissertation.

Dr. Gabriela Arévalo for presenting me with an interesting subject and granting me the freedom to work on it at my own pace.

Peter Ebraert for introducing me to the EMOOSE program and helping me fill out the necessary paperwork.

All my friends of the EMOOSE program, with who I had a wonderful time in Nantes.

Laura Sánchez for all the loving support, especially during the hardest phases of the program.

My parents for allowing me to start an interesting year of studies abroad.

Contents

List of Figures

1 Introduction

One of the major factors for the invention of the object-oriented programming paradigm, was to remove some flaws of the procedural programming paradigm. About forty-five years after its invention, twenty years after it started becoming mainstream, it is well-known that compared to its ancestor, this paradigm increases *reusability* and *maintainability*. However, *understandability* is another crucial factor to consider that encompasses the other two. As shown in several case studies [WH92], understanding the systems takes more than half of the time during software maintenance. Thus, we can see that architecture recovery is a complex and time-consuming process.

When a software engineer has a first contact with an object-oriented application, in order to understand the system, he needs to get a mental image of the system. He needs to recover what the original conceptual architecture of the system was. For this purpose, the object-oriented programming paradigm is already more powerful compared to procedural code, because it offers *encapsulation* (grouping related functionality and data into classes) and *class inheritance* as mechanisms to explicitly embed these parts of the conceptual design into the code.

When considering applications with huge amounts of classes, the level of abstraction available at code-level might not match the conceptual level of abstraction. This is also true when while during the implementation or maintaining phase, proper object-oriented programming rules were ignored. The combination of those two problems is even worse. These facts, in spite of the promising decomposition implied by the paradigm, decrease the understandability of object-oriented systems.

In order to draw the burden of recovering the original conceptual architecture, away from the maintainer, it is useful to investigate ways of automating this process.

1.1 Problem Statement

The idea of automating the recovery of the architecture of software has already been explored quite extensively for the procedural programming paradigm. This is logical since, up until a few years ago, most of the *legacy systems* where developed using that paradigm. In his PhD thesis, Rainer Koschke [Kos02] has worked on a deep analysis of several representative techniques. Additionally, he presented ways of combining those techniques and described a way of involving the user into the process, in order to increase the effectiveness of (semi-)automatic architecture recovery.

However, nowadays a fair share of systems are built using object-oriented paradigm. Unfortunately, most of the techniques resulting from research conducted to architecture recovery are not up to the task of being applied to this type of systems.

1.2 Proposed Solution

One solution in the quest for automating architecture recovery, could be to look for new ways of recovering implicit architectural information from a system. This path was taken by related research such as [WSY$^+$06, LSLW05, LLSW03, KP96, eAPS00].

Yet we believe that when conducting research, before coming up with new ideas, it is important to investigate the adaptability of previous research to a new environment. In this thesis we investigate how adaptable some of the already existing architecture recovery techniques are to the object-oriented environment. Especially if we look at techniques defined for procedural programming, we see that the setting and goal are actually quite similar. Koschke [Kos02] already remarked that the difference between detecting atomic components and bigger entities is just a matter of granularity.

Therefore, we focus our research on the techniques described in the thesis of Koschke [Kos02], which are defined for procedural programming languages. And specifically, we investigate if they are translatable as such that they are applicable to object-oriented and dynamically-typed systems.

We restrict our research to the adaptation of the techniques by minimizing the possible amount of changes to the original techniques. The goal is to clearly draw the line between *adapted* techniques, and techniques which are merely *based on* the ideas behind another technique.

Since object-oriented paradigm has the advantage of being better structured by nature, we will extend this research by investigating up to what level the object-oriented structure can be leveraged to complement the adaptations of the original techniques. The limitation we place on this part of our research is that this may not result in new *heuristics*.

1.3 Outline

This report is organized as follows: Chapter 2 defines the *context* of this thesis.

Chapter 3 explains what the global changes are when adapting the *automatic techniques* described in [Kos02], and reports the specific adaptations for every technique we adapted.

Chapter 4 shows how the techniques, adapted in Chapter 3, are combined in order to increase their effectiveness.

Chapter 5 presents an extended interactive method in two parts. Section 5.1 explains how techniques are combined and presented in an optimal way. And Section 5.2 describes techniques to evaluate candidate components.

Chapter 6 validates our adapted techniques by applying them to a “real-world” application.

Finally, we conclude the findings of this thesis in Chapter 7.

2
Context

This chapter explains the used terminology and provides an introduction to the research fields focused on this thesis.

2.1 Reengineering

When a system must be changed, the software engineer must follow a *reengineering* process. Cross and Chikofsky [CI90] define this process as

> the examination and the alteration of a subject system to reconstitute it in a new form and the subsequent implementation of a new form.

This definition comprises two main activities, namely *examination* and *alteration* of a system. More formally, these activities are defined as follows.

2.1.1 Examination

Reverse Engineering is the process of analysing a subject system to (i) identify the system's components and their relationships and (ii) create representations of the system in another form or at a higher level of abstraction.

2.1.2 Alteration

For the alteration phase, there are two different types of activities to be applied. Mostly, the higher-level view is firstly *restructured* in some way, and then *forward* *engineering* is applied.

Restructuring is the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics). Restructuring is often used as a form of preventive maintenance to improve the physical state of the subject system with respect to some preferred standard.

Forward Engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.

2.1.3 Procedural Architecture Recovery

Koschke [Kos02] stated that there was still a debate on what could actually be considered part of the *software architecture*, but that most agree that it should at least include *components* and *connectors*. Since the research was done in procedural environment, the most basic form of such components were *atomic components*, which are comparable to the concept of prototypes[1]. These atomic components are defined in a level of abstraction which is higher than the level of abstraction which is actually available in procedural code.

2.1.4 Object-Oriented Architectural Recovery

Even while most of the research regarding to architecture recovery, up until now has been conducted in a procedural setting, in order to understand object-oriented systems, it is equally necessary to recover their architectural information.

Architectural Unit

If we adapt the definition of recovering architectural information of section 2.1.3 to object-oriented architecture recovery, this results in the detection of components similar to *software components*.

Problems in Object-Oriented Systems

As defined above, when recovering architectural information in an object-oriented setting, the goal is to detect and expose different dependencies between software artifacts of components. And this by only applying statical analysis to the subject system's code. Even more than when recovering architectural information for

[1]For more information on prototypes, see [TM01].

procedural code, this is not a trivial task because in most cases the code contain implicit dependencies [Are05].

Dependency. An object A depends upon another object B, if it is possible that a change to B implies that A is affected or also needs to be changed, i.e., dependency between a *client* and a *server*.

Explicit Dependency. A dependency between two or more objects is explicit when it is precisely and clearly expressed without ambiguity in the source code, i.e., definition of a direct subclass (in Smalltalk using the keyword `superclass` or in Java using the keyword `extends`).

Implicit dependency. A dependency between two or more objects is implicit when it is implied by the source code though is not directly expressed, i.e., chain of superclasses of a newly defined class

In our approach, we will focus on modeling architecture recovery solely on explicit dependencies. By applying architecture recovery clustering techniques, it is expected that resulting clusters expose implicit dependencies between enclosing classes.

2.2 Component-Based Software Development

In component based software development (CBSD), software components are the basic units of reuse, which provide a relatively coarse-grained functionality. A software component described in an object-oriented setting, typically consists of one or more related classes which in collaborate order to carry out system operations [HC01]. In order to build systems using CBSD, several software components are to be chosen and their interfaces to be combined with glue code.

While the granularity of componentware and object-oriented programming differ, their goals are very similar: facilitate and thereby increase the reuse of software, in order to make it more reliable and less expensive [HRR98].

2.2.1 Our Definition of Object-Oriented Components

Software components are considered to be *stand-alone* pieces of software, so-called *black-boxes*, which interact with each other using (different types of) interfaces. Classically, these interfaces are defined as such that they only allow communication between components by means of message passing. However, if we are to build software using object-oriented components, it makes sense to extend the specification of possible interfaces of components as such that they at least match the flexibility of the specification of possible interfaces of classes. For this reason, we identify two types of *provided interface elements*:

Message Interface part represents what classically is considered as a valid part of a component interface. Components using a given component communicate with this type of interface elements, by means of message passing. Message passing between interfaces of components does not necessarily equal message passing between actual internal classes, but is a possible implementation, as will be the case in the systems we inspect[2].

Subclass Interface part represents a part of the interface which makes the component "subcomponentable". Those elements are generally classes of which the interface is provided as part of the interface of the component, so that other components can subclass from them. This allows components to be placed hierarchically[3], or the other way around, allows us to split class hierarchies in components. This is a needed feature if we are to decompose full object-oriented systems, including system libraries, into components. This also allows us to reuse tools originally designed to evaluate resulting components, to evaluate already available decompositions into *packages* and *namespaces*, which are also possibly defined as such that they break class hierarchies.

2.2.2 Identifying Components at Design Level

While CBSD paves the way to better decomposition which improves understandability, maintainability and reuse, because of the level of abstraction, it is not always easy to reason about software in terms of software components. For this reason, several approaches have been presented in [JCIR01, CD00, STS99, LYC$^+$99, KC04] which help software developers into identifying software components, given that object oriented models for domain applications are available.

2.2.3 Identifying Components at Implementation Level

The work presented in this thesis, can also be considered to belong in the previously defined field. However, we go out from the idea that we do not have

[2]The systems we inspect "mimick" message passing between components by passing messages between classes, since we only investigate systems built in object-oriented languages which are not equipped with special component-oriented constructs.

[3]Placing components hierarchically is defined in the same line as class hierarchies, not how *hierarchical components* are defined in projects such as *SOFA* [BHP06] and *Fractal* [BCL$^+$06], where bigger reusable components are build as compositions of smaller ones.

object-oriented designs, but only object-oriented implementations at hand for detecting components. Our goal is then to reverse engineer such systems and to generate high-level views in which the implicit decompositions of the systems into software components are made explicit.

2.3 Related Research

Apart from the related research listed in chapter 1, there have been two other studies more closely related to our research.

Trifu [Tri01] investigated how *Dominance Analysis* as well as *Similarity Clustering* could be applied to object oriented code. His ideas are based on [Kos02]. In his work, the techniques are defined using all available information about object-oriented programs. This includes *invocation candidates*. He presents a four stage methodology for clustering consisting of *analysis*, *compaction*, *clustering* and *result interpretation*.

In [BT04], a incremental approach to component detection is presented. This approach combines an adapted version of *Similarity Clustering* from [Tri01] with logic meta-programming which is used to detect design patterns.

2.4 Summary

Our research is focused on *reverse engineering* and *restructuring*. The goal is to recover architectural information of object-oriented systems, relying only on *explicit dependencies*. Our approach is based on similar research conducted by R. Koschke in a procedural setting, where he used clustering algorithms to detect *atomic components*. We adapt those algorithms to detect *software components* in an object-oriented setting, which are expected to expose *implicit dependencies* between enclosed classes.

3 Adaptation of Automatic Techniques

The detection of atomic components in procedural code was the main goal of the algorithms in the PhD thesis of Koschke [Kos02]. Those techniques generally intent to optimize the cohesion/coupling ratio and maximize encapsulation in the components.

The main drawback of these techniques is that they can only be applied to procedural code, while the idea of software components in object-oriented component is quite similar.

In this chapter we will discuss how to change or extend some of the atomic clustering algorithms, so that they can be applied to object-oriented code, in order to find meaningful clusters of classes which help maintainers to understand object-oriented applications.

3.1 Shifting Focus

When we adapt component clustering techniques from procedural code to object-oriented code, in most cases the idea stays quite similar, but the type of subject systems change. This allows us to specify a subset of general changes which apply to all techniques.

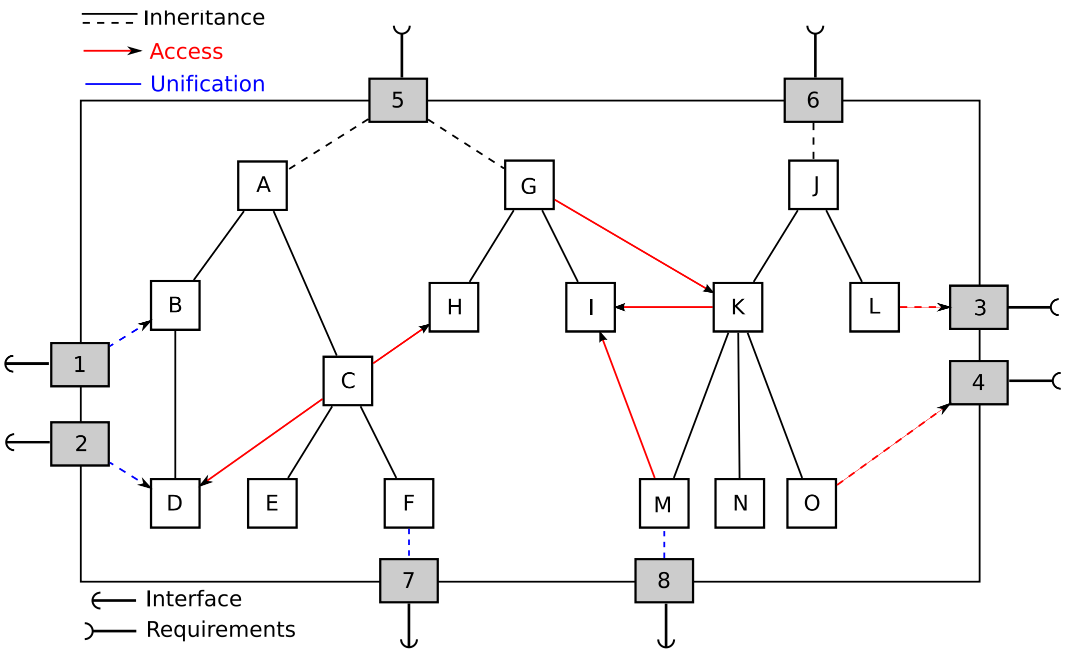


Figure 3.1: An Object-Oriented Software Component

3.1.1 Global Objects and Procedures

The main difference in the clustering data is that the difference between encapsulated data and the interface of the resulting component fades away, by shifting the focus from global objects and procedures to classes. As opposed to the original ideal situation, where global objects were totally encapsulated in resulting atomic components, classes have a double role in software components. They can represent (a part of) the interface as well solely internally used elements. It can be envisioned that adapted techniques based on the idea of encapsulation will be cluttered with evermore false positives, due to the unification of encapsulated global objects with interface classes. For example, this is the case for classes B, D, L and O in figure 3.1.

3.1.2 Referenced Objects and Procedure Calls

Most of the original techniques used references to global objects and procedure calls as information for grouping elements together. Ignoring the availability of function-pointers, these *links* in procedural code are mostly statical. This fact validates their use.

When adapting this kind of techniques, in a class-based environment, both references as well as method calls are available. However, especially for a dynamically typed, class-based programming language, information about method calls is not as static. This mostly results from the use of *virtual tables*. Where procedure calls where mostly statically computable, statical analysis of polymorphic methods calls result in sets of *invocation candidates*, most likely containing several false positives.

Taking this into account, when adapting techniques using object references and/or procedure calls as clustering information, one is presented with a choice. Either we choose to use the *invocation candidates* as clustering information, already cluttering the clustering information with false positives, which will result in more false clusters. Or we decide to narrow the information down to only using statical references to classes. In the section 3.1 we have mentioned that adapting techniques related to encapsulation will already inevitably result in more false positives. For this reason, we are in favour of having more true negatives as opposed to having even more false positives, therefore we only use statical references as narrowed clustering information. This follows the approach we set out in section 2.1.4, where we defined to model our approach by only using *explicit dependencies* between classes.

An advantage of working in a class-based environment, is the explicit presence (at least for well-designed systems) of relations between classes. This allows us to define the function *referred-by*(c), or the set of classes that a class c refers to, as the union of the set of references of the methods defined in the class (thus excluding inherited methods), with the set of direct superclasses. Excluding inherited methods allows us to keep the difference between different classes in hierarchies. Including direct superclasses on the other hand, allows us to include the available hierarchical information. As we will see, this drastically improves the results of algorithms such as the *Strongly Connected Components* algorithm, defined in section 3.2.4.

$$\textit{referred-by}(c) = \textit{direct superclasses}(c) \cup \textit{class references}(c) \tag{3.1}$$

$$\textit{refers-to}(c) = \textit{referred-by}^{-1}(c) \tag{3.2}$$

For example, in figure 3.1, *referred-by*$(M) = \{I, K\}$ and
refers-to$(K) = \{G, M, N, O\}$

3.2 Adapted Heuristics

In this section we give a detailed description of the adaptation of automatic component detection algorithms. Each technique is described using a small *pattern* containing three sections:

Original Clustering Criterion explains which was the goal of technique described in its original source. If no source is explicitly mentioned, the thesis of Koschke [Kos02] is the implied source.

Revisited Clustering Criterion describes how we adapt the original clustering criterion to one applicable to object-oriented, dynamically-typed systems. This is a superficial definition which deviates as little as possible from the original criterion.

Sidenotes details extra features related to the adaptation of the clustering algorithm. It highlights possible problems and/or deviations from the original techniques.

3.2.1 Global Object Reference Heuristic

Original Clustering Criterion. Global objects and all the routines that reference these objects, regardless of where they are declared, are grouped together.

Revisited Clustering Criterion. Classes, all classes which reference these classes in their methods, regardless of where they are declared, and all direct subclasses, are grouped together.

Sidenotes. As for the original version, widely used classes may cause large parts of the system to collapse into one big cluster. Yeh et al. proposed to exclude frequently used objects from the analysis to avoid this unwanted effect. Even more in the adapted version, excluding widely used classes will improve the performance of the technique, since unlike the use of global objects, the use of library classes is not so exceptional.

While for the original version, after removing widely used global objects as clustering data, good results are to be expected, the adapted version suffers from more false positives due to the unification of global objects with interface classes. For example, in figure 3.1, suppose that the system is fully decomposed into software components, requirements 1 and 2 represent classes external to the given software component. Since they are used by the given software component, this means that those classes are part of the interface of another software component. If those classes would now be considered *global data objects* for the *Global Object Reference* heuristic, these classes would be included in the given software component since they are referred to by classes A, G and J.

3.2.2 Same Module Heuristic

In well designed software, the system is properly decomposed and all modules contain single components. When we count on good design, we can group all declarations of a module together in a component. This is the underlying clustering criterion of this technique.

Original Clustering Criterion. All related subprograms, user-defined data types, and global objects that are declared in the same module are grouped together. A subprogram is related to a data type when the data type occurs in the signature of the subprogram. Likewise, a subprogram is related to an object when the subprogram references the object.

Revisited Clustering Criterion. Since we expect the base system to already use multiple different ways of grouping classes together, we split the original technique into three different heuristics. The first yields clusters of classes per *namespace*, the second per *package* and the last one per *subhierarchy*.

Sidenotes. In the original technique, the purpose was to find single atomic components in modules. Adapting that technique to cluster namespaces and packages, can also have the effect of finding at least parts of software components in those groups of elements. The subhierarchy adaptation will be useful if the chosen superclass belongs to a software component and none of its subclasses are part of the subclassed interface.

3.2.3 Delta-IC

High cohesion in a software component *S*, means that all of the classes of *S* refer to many classes in *S*. Low coupling implies that those classes refer only to very few classes which do not belong to *S*, and that only very few classes from outside of *S* refer to classes in *S*.

The *Delta-IC* technique is a step in that way. It first clusters elements together, according to a specific usage pattern. Then all resulting clusters are evaluated by a connectivity metric, and accepted or rejected resulting on their relation to a given threshold.

Original Clustering Criterion. A candidate for a given subprogram *S* is *candidate-cluster*(S) where $\Delta IC(S) > \Theta$.

$$\textit{closely-related subprograms}(S) = \bigcup_{e \in \textit{referred-by}(S)} \{F | F \in \textit{refers-to}(e) \wedge \textit{referred-by}(F) \subseteq \textit{referred-by}(S)\} \tag{3.3}$$

$$\textit{candidate-cluster}(S) = \textit{closely-related subprograms}(S) \cup \textit{referred-by}(S) \tag{3.4}$$

Revisited Clustering Criterion. The revisited clustering criterion equals the original one. We only adjust the function candidate-cluster to take classes as input and to have clusters of classes as output. The definition of the revisited formula complies with the shift of focus presented in section 3.1.2, such that *refers-to*(e) represents a set of classes which refer to (reference or subclass the given class *c*).

$$\text{closely-related classes } (C) = \bigcup_{e \in \textit{referred-by}(C)} \{F | F \in \textit{refers-to}(e) \wedge \textit{referred-by}(F) \subseteq \textit{referred-by}(C)\} \tag{3.5}$$

$$\textit{candidate-cluster}(C) = \textit{closely-related classes}(C) \cup \textit{referred-by}(C) \tag{3.6}$$

Koschke [Kos02] suggested to use a different metric than the one that was originally presented by Canfora et al. because of the fact that external connectivity was not correctly represented in that metric. For the same reason, we also use the *Internal and External Connectivity* metric proposed by Koschke.

$$\Delta IC(C) = \textit{connectivity}(C) \tag{3.7}$$

$$\textit{connectivity}(C) = \frac{a \times \textit{IntC}(C) - \textit{ExtC}(C) + 1}{a + 1} \tag{3.8}$$

$$\textit{IntC}(C) = \frac{1}{\mid \textit{referred-to}(C) \mid} \times \sum_{e \in \textit{referred-to}(C)} \frac{\textit{refers-to}(e) \cap \textit{referring}(C)}{\textit{referring}(C)} \tag{3.9}$$

$$\textit{ExtC}(C) = \frac{1}{\mid \textit{referred-to}(C) \mid} \times \sum_{e \in \textit{referred-to}(C)} \frac{\textit{refers-to}(e) \setminus \textit{referring}(C)}{\textit{refers-to}(e)} \tag{3.10}$$

Sidenotes. Since referring entities and referred entities in our case are represented by the same type of entities, it is possible that the referred entities overlap the closely related classes. This is the reason why the definitions of *IntC* and *ExtC* are not as straightforward as in the thesis of Koschke. We need the following definitions to extract the original values:

$$\textit{extract}(C, f) = \{F | \{E | C = \textit{candidate-cluster}(E) \wedge F = f(E)\}\} \tag{3.11}$$

$$\textit{referred-to}(C) = \textit{extract}(C, \textit{referred-by}) \tag{3.12}$$

$$\textit{referring}(C) = \textit{extract}(C, \textit{closely-related classes}) \tag{3.13}$$

Of course, at implementation level these intermediate results, calculated to build the candidate cluster, can be reused at the moment that we apply the metric.

3.2.4 Strongly Connected Components

Cycles in the graph of classes connected by the *referred-by* relation form (parts of) software components, because none of the classes can be omitted without losing a piece of information for the understanding of the referring classes in the software component. These cycles correspond to the notion of strongly connected subgraphs in graph theory.

Original Clustering Criterion. All strongly connected components in the call graph form an atomic component.

Revisited Clustering Criterion. All strongly connected components in the *referred-by* graph form a software component.

Sidenotes. Mutually recursive subprograms in procedural code are less common, and are assured to belong together. They are less common since procedures generally refer to (call) only a small amount of other procedures.

By refocussing procedures to sets of methods of classes, this type of relationship gets less uncommon. This partly results from class extensions to classes which are actually data classes, in other words, *extended data classes*. This type of classes are generally referred to by a big part of the other classes in the system. On their own, they also refer to a big part of the other classes. Systems containing such classes will collapse in big software components based on strongly connected components, with those classes as “center class”. In this case it is important that the user is able to remove certain false *referred-by* relations in order to decouple such strongly connected graphs into possibly multiple new strongly connected subgraphs.

Because of the definition of referred-by as stated in section 3.1.2, by incorporating inheritance information, we also find cycles between classes where the referred class does not necessarily link back to the referring class, but also when it refers to a superclass of the referring class. Secondly, since instead of incorporating this information as the set of references of all methods (of the class and all inherited methods), we use the set of references of the local methods *and* outgoing inheritance relationships[1], we do not only find more cycles between classes which refer to one another, but also which parts of hierarchies are responsible for the cycles.

[1]**Outgoing inheritance relationships:** the set of all direct superclasses

3.2.5 Dominance Analysis

Software components often have local classes (or hierarchies) which offer local services to classes that constitute the interface of the components. Since these basic service classes are local to software components, they are an essential part of them and the software components can not be understood without them. Local in this context means that they are only referred to by classes in the software component. It does not mean that they are local in the sense of nested scopes (nested classes); quite the opposite: Because they may be used by several other classes in the software components, they must be visible to all of them. This kind of local routines can be detected by means of dominance analysis. Locality in the mentioned sense can be viewed as a dominance relation in graph theory.

Original Clustering Criterion. Cimitile and Visaggio (1995) propose to apply dominance analysis to call graphs to identify candidates for reusable modules. First cycles (i.e. strongly connected components) are collapsed before dominance analysis is applied.

Revisited Clustering Criterion. Dominance analysis is applied to referred-by graphs, in which strongly connected components are collapsed, to identify candidates for reusable software components.

Sidenotes. Collapsing previously found clusters, as was described by Koschke [Kos02], improves the results of dominance analysis. If we do not collapse already found clusters first, it is possible, as shown in figure 3.2 that two classes B and C are found to be part of a cluster, both of which are dominating a class A. If those two classes B and C are part of the interface of the found cluster, it is possible that they are both referred to by a class D. Applying dominance analysis without collapsing B and C into one node, as happens now, results in A being dominated by D, since as well B and C are different paths to go from D to A. Already collapsing B and C presents the user with the correct result, namely that A is dominated by the interface (B and C) of an already found cluster.

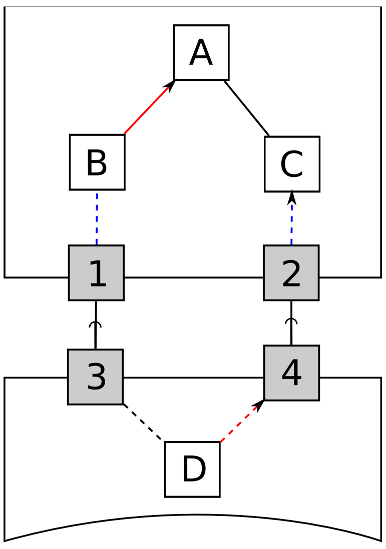


Figure 3.2: Two Components

Koschke defined his application of the dominance analysis technique as such that collapsed components were necessary. It is defined as such that all elements dominated by a component are added to that component. In his definition, dominance analysis can only be used as an additional technique. We however, believe that results from dominance analysis can already present interesting information (however limited), without using already defined components as basis.

By using the algorithm in this way, the adapted version suffers from false positives due to the solely local use of *libraries*. For this reason, it is interesting to present a dominance tree to the user as such that he can select from which root the elements actually belong to software components, but such that he can choose from which subnode the dominated subtree actually represents an external[2] library. The resulting cluster would then include the tree starting at the included root, cutting the subtrees which are out of the software component's scope. This idea is similar to the way of browsing the result of *similarity clustering*, presented in section 3.2.6.

Another difference we consider from Koschke, is that our programs can have different *entry points*. An entry point is a piece of code where a program might start. If we broaden this point of view, all pieces of software which are not explicitly referred to, can start from any point, or from none. Especially for dynamic languages like Smalltalk, where it is for example possible to instantiate all classes defined in a certain package, it is important that clustering algorithms take this possibility into account. For that reason, the adapted version of the technique adds a *supernode* to the directed graph of all relations, which links to all entry points of the subject system.

3.2.6 Similarity Clustering

If two classes use several of the same set of classes, as well as when they are used by several similar classes, they are likely to be sharing a significant amount of design information. In that case they are good candidates for placing in the same module. The similarity clustering algorithm defined in the thesis of Koschke is an extension of *Schwanke's Arch Approach*.

Original Clustering Criterion. Before starting the clustering algorithm, all procedures are placed in their own group. In each iteration, the two most similar groups are combined using the similarity metric for groups. This happens until no new group can be found which has higher similarity than a given threshold.

[2]External in this situation refers to *not belonging to the software component*, not necessarily to *outside of the scope of the subject system*. If the class is a part of the subject system, but external, it belongs to another software component of the system

The original similarity between groups, proposed by Schwanke (1991), was defined as **maximal individual similarity** (also known as *single linkage*):

$$GSim(A,B) = max(Sim(a,b)) \quad \forall a \in A, b \in B \tag{3.14}$$
$$Sim(A,B) = \frac{W(Common(A,B)) + k \times Linked(A,B)}{n + W(Common(A,B)) + d \times W(Distinct(A,B))} \tag{3.15}$$

In the definition of $Sim(A,B)$, $Common(A,B)$ represents the amount of common features of A and B. $Linked(A,B)$ returns 1 if A calls B or B calls A, otherwise it returns 0. $Distinct(A,B)$ represents the amount of distinct features of A and B. Features of a procedure in this scope means all types of non-local references. W is a weighting function which ensures that more widely used elements have less impact on the total result, and that elements with a low occurrence rate have the opposite effect.

$$Common(A,B) = features(A) \cap features(B) \tag{3.16}$$
$$Distinct(A,B) = features(A) \oplus features(B) \tag{3.17}$$
$$W(features) = \sum_{f \in features} w_f \tag{3.18}$$
$$w_f = -\ln Probability(f) \tag{3.19}$$
$$Probability(f) = \frac{occ(f)}{occ(type(f))} \tag{3.20}$$

In formula 3.17, $operator \oplus$ denotes the symmetric difference for sets, and the $Probability(f)$ function represents the *Shannon information content* [Sha53] from information theory.

Koschke extended Schwanke's approach as such that the similarity between groups was defined as a hierarchical clustering algorithm with *average linkage*.

$$GSim(A,B) = \frac{\sum_{(a \in A, b \in B)} Sim(a,b))}{\mid A \mid \times \mid B \mid} \tag{3.21}$$

Secondly, the similarity metric between two procedures was extended to the following definition:

$$\begin{aligned} Sim(A,B) = {} & x1 \cdot Sim_{Indirect}(A,B) \\ & + x2 \cdot Sim_{Direct}(A,B) \\ & + x3 \cdot Sim_{Informal}(A,B) \end{aligned} \tag{3.22}$$

Here there are two differences to the original algorithm. The definition of $Sim_{Indirect}$ has been changed to take overlapping references by different types of connections into account. For this reason the function $Common(A, B)$ was separated into $Common_{eq}(A, B)$ which represents common features by a same type of reference, and $Common_{ne}(A, B)$, which represents common features which are bound to A in a different way from how they are bound to B. By separating common features, different weights (different importance) can be assigned to different types of common features.

Secondly the $Sim_{Informal}$ is added, in order to take informal information about the procedures into account. This informal information consists of information about the words used in the definition of the procedure, as well as words used in the name of the file where the procedure was defined.

Koschke suggested to keep the history of joins of groups and present this as a result to the user. In this way, the algorithm can join until there is just one group left. The user can then inspect the resulting tree and only accept candidate clusters at subtrees of nodes where the join is doubtful.

Revisited Clustering Criterion. We adopt the change of definition of $Gsim(A, B)$ and $Sim(A, B)$ by Koschke. However, since we only consider one type of relation between classes, the extended definition of $Sim_{Indirect}$ by Koschke can be replaced by the easier original definition by Schwanke[3].

As for the informal information of classes, we define it as a combination of four different relations found in a class-based setting.

$$Ratio(A, B, f) = x_f \cdot \frac{W_f(Common_f(A, B))}{W_f(Common_f(A, B)) + W_f(Distinct_f(A, B)} \tag{3.23}$$

$$\begin{aligned} Sim_{Informal}(A, B) = {} & Ratio(A, B, MethodNames) \\ & + Ratio(A, B, AttributeNames) \\ & + Ratio(A, B, Superclasses) \\ & + Ratio(A, B, Surroundings) \end{aligned} \tag{3.24}$$

In which $Surroundings$ represents all information available about the scope in which the classes are defined. This is similar to what Koschke did with filenames. In our case, the available types of information are the namespace and package in which the class is defined.

Notice that for any of the different types of features which classes can have, a different weighting function W_{type} is used, as well as different functions $Common_{type}$ and $Distinct_{type}$. These two last use $features_{type}(Class)$ as base function. In the same way, the functions for $Sim_{Indirect}$ can be defined in function of $Common_{referred\text{-}by, refers\text{-}to}$ and $Distinct_{referred\text{-}by, refers\text{-}to}$.

[3]Actually there are two different types of relations which are joined in the definition of referred-by, *referenced by* and *direct superclasses*. It might be interesting to investigate what would change if we make a difference between those types of relations. This is suggested as future work.

Sidenotes. The result of applying a hierarchical clustering algorithm to the distances between elements, is a **dendrogram**, i.e. a binary tree, whose leaves are clustered classes and whose nodes represent the union of the two sub-clusters. The further nodes are from a root, the higher the similarity-level of the cluster it represents.

A dendrogram is useful information since it shows the order in which entities are clustered, and the respective most similar entities. Hence the result of *Similarity Clustering* should be shown as a dendrogram to the user. A minor aid to the user, in order to go through the resulting information as fast as possible, would be to cut unnecessary information from the tree. As such our method removes leaves (clusters with only one element, which are not considered *real* clusters), and does not show direct subnodes which have the same similarity as their supernode. This can especially be useful when an amount of entities all lay at the same distance from each other. It is of no importance to the user, which of the entities are joined before others are added. This is probably merely randomly decided by the algorithm, based on internal states. As such, subnodes whose parent have the same similarity level are discarded, and their children are recursively added to their parents. This results in a tree which is not necessarily binary.

One of the main disadvantages of using this technique, is the time-complexity needed to calculate the distances between elements. In the worst case, all classes refer to all classes, which implies that calculation of second degree neighbours for m classes, has a cost of $O(m^3)$. Koschke described that, since procedures generally only have a small amount of neighbouring procedures, the *real* complexity was a lot lower. This reduces extremely the time needed to compute the similarity matrix. Unfortunately, in a class-based setting, we do not necessarily share this property. As was already stated in section 3.2.4, relations between classes are, because of the use of certain object-oriented patterns, less uncommon. *Extended data classes* have a significant negative impact on the complexity. As such, it is important that the user can identify and ignore unnecessary relations going to and coming from this type of classes, before applying the algorithm.

In addition to the advantage related to time-complexity, also the results of the algorithm will be less cluttered after removing relations between objects which are only present in the code to fulfill certain object-oriented tasks.

3.3 Summary

In this chapter we have presented how to adapt six different atomic component recovery algorithms for procedural languages, in order to find meaningful clusters of classes in dynamically-typed, class-based languages. The techniques are only based on structural information.

The atomic component recovery techniques we have chosen as basis for our research, are techniques which are, at least to some extent, adaptable such that they only require pure object-oriented information about their subject system. We have made a selection of algorithms which were defined as automatic component recovery techniques in the thesis of Koschke, which are not fully based on the availability of a static typesystem.

These two requirements ruled out the adaptation of the *Internal Access / Non-Abstract Usage Heuristic*, which is based on internal accesses of objects and types. It also ruled out the adaptation of the *Type-Based Cohesion* and *Part Type* heuristics, which are defined based upon the idea of the availability of static typing.

The techniques we have adapted can be roughly classified and adapted as follows:

- **Connection-based** approaches cluster entities based on a specific set of direct relationships between entities to be grouped. Of the adapted algorithms, *Global Object Reference* and *Same Module* fall into this category. The only types of connections between entities we consider in the adapted algorithms, are class hierarchy relations and direct references in bodies of methods.

- **Metric-based** approaches cluster entities based on a metric using an iterative approach. The *Similarity Clustering* approach falls under this category. This approach is partly based on connections too, but the difference lies in the way the used metric can fine-tune the importance given to certain relations. For this heuristic, the metric is adapted such that, instead of taking available procedural features into account, it now defined in terms of the available object-oriented features.

  Koschke also classified *Delta-IC* in this category since, next to its underlying connection-based clustering heuristic, it also uses a metric to filter out non-relevant clusters. However, since all information used for the underlying heuristic, as well as for the metric, are based on global entity usage, this approach can be adapted in a similar way as the connection-based heuristics.

- **Graph-based** approaches derive clusters from a graph by means of graph-theoretic analysis. The *Strongly Connected Components* and *Dominance Analysis* heuristics belong to this category. The difference between these approaches and connection-based approaches lies in the way the algorithm handles connections. This implies that, since we are only changing types of nodes and edges, these algorithms can be adapted as straightforward as the connection-based heuristics.

As we can see, all techniques we adapted, except for the *Similarity Clustering* technique, can be adapted by using the referred-by, refers-to and class hierarchy relations defined on classes as only types of connections. *Similarity Clustering* is more complex and needs to be adapted separately.

4 Combined and Incremental Techniques

In the previous chapter, we have presented a set of automatic clustering techniques, adapted from atomic component clustering techniques. Already for the original techniques, it was clear that none of the techniques resulted in components up to the standards of components detected by humans. Even more, because of the adaptation problems stated in the previous chapter, this is true for the adaptation of the techniques which detect software components in a class-based setting.

There are two different alternatives which improve the research related to detecting software components. A first one is building more advanced, new heuristics which are not necessarily based on the techniques presented in the previous chapter. A second alternative is to look for an approach to optimize the results of existing techniques.

We chose the second alternative due to several reasons. Because it makes more sense to first explore how we can improve the results of existing techniques, before new techniques are tackled. Secondly because our research is partly centered around the question to which extent we can successfully adapt existing atomic clustering techniques to detect software components.

In this chapter, we show how software component detection techniques are combined in order to improve results, and how object-oriented information about the subject system is leveraged to complete candidate components.

4.1 Combining Results

Applying clustering algorithms to a set of classes will, obviously, result in clusters of classes. Applying clustering algorithms to procedural code however, results in different types of clusters, based on the entities which are affected by the clustering algorithm in question. This makes the combination of techniques applied to procedural code more complex than just combinations on sets. However, fortunately, since we are only clustering classes, normal set operators will do just fine for most of the possible ways of combining techniques. This results in the fact that we can reuse operators such as the *union*, *intersection* and *difference* operators defined for sets, for clusters.

4.1.1 Extending Results

A first way to define the composition $T_2(T_1(Classes))$ of two techniques T_1 and T_2, is to define it as a such that the candidate clusters provided by T_1 are used as valid clusters which are extended by T_2.

$$Composition(T_1, T_2, C) = \textit{MergeOverlapping}(T_1(C) \cup T_2(C)) \tag{4.1}$$

In which *MergeOverlapping* is defined as a function which iteratively combines all overlapping sets until all remaining sets are disjunct.

We define it this way, because we go out from the position that approved candidate clusters imply an *actually belong together* relation between the classes in the cluster. It is possible that the elements of two distinct candidate clusters found by one technique, also belong together. Since a second technique has found that those elements also belong together, their respective clusters are combined into one cluster. In other words, if T_1 has found that $\{A, B\}$ and $\{C, D\}$ belong together, and T_2 has found that $\{B, C\}$ belong together, then it is only logical that $\{A, B, C, D\}$ belong together.

As such, the *space of components* is defined a *set of disjoint sets*. For convenience, we can define the data structure used to represent the space of components in a way that it always enforces this restriction, in other words, a *disjoint-sets datastracture* [THCS01]. Like this, when we now combine techniques in the explained way, we just have to add all their resulting clusters to a *space of components* which automatically handles overlapping.

4.1.2 Refining Results

As a second way of combining techniques, we define it as such that T_2 refines the results of T_1 by looking for clusters in the candidates resulting from applying T_1

to a set of classes. The idea of refining results, stems from the fact that techniques with several false positives result in clusters which are too big. The original technique has found that the classes in a big cluster probably belong together, but it is classified as a false positive since it is not refined enough to define a software component.

There are two ways of defining a refining composition. In the first one, the same information about the system is provided to both techniques. Then the resulting clusters are intersections of all pairs of clusters with one cluster from the results of T_1 and one cluster from the results of T_2.

$$Composition(T_1, T_2, C) = \bigcup_{C_1 \in T_1(C), C_2 \in T_2(C)} C_1 \cap C_2 \tag{4.2}$$

The result of this definition can then be filtered so only real clusters[1] remain.

In the second type of refining, we provide local information of all clusters found by T_1, one by one to T_2.

$$Composition(T_1, T_2, Classes) = \bigcup_{Cluster \in T_1(Classes)} T_2(Cluster) \tag{4.3}$$

In the previous definition we can define $T_2(Cluster)$ as such that we consider the availability of all the relations between classes in $Cluster$ as well as relations from $Cluster$ to $Classes$. Or we only consider how classes inside the cluster react to each other, discarding relations from $Cluster$ to $Classes$. In this way, refining results of a technique by the same technique, might result in clusters inside a previous candidate cluster, since all side-information is discarded. Actually, the first option is similar to the first way of refining results, but defined in a way that needs more computation. Also when the second technique differs from the first one, by implementing the combination this way, the second technique will only receive a limited amount of information (limited by the first technique). If we would not do this, the second technique would produce more true negatives. That is why for the second way of refining clusters, we choose the second definition.

4.2 Collapsing Results

When T_2 is a graph-based technique, as already stated in section 3.2.5, it is useful to collapse previously found clusters into single nodes, before applying dominance analysis. Koschke stated that it might also be useful to collapse clusters before applying *Strongly Connected Component* detection, in order to find which

[1]Real clusters are clusters with at least two elements

components can not live without each other. This is no different for strongly connected software components. However, it might be useful to first apply the *strongly connected base components* algorithm to detect and remove *false references* as described in section 3.2.4, before applying the *strongly connected collapsed components* view, where such relations might be hidden.

Implementing collapsed techniques is quite straightforward in a object-oriented language. Instead of feeding the list of classes to the algorithms, classes contained in clusters are replaced by their enclosing clusters. Clusters then have to be extended as such that they answer to the same messages as normal classes. Actually, this is not only useful for this purpose, but will also be useful for section 4.4.1. Secondly, classes resulting from message sends which are contained in clusters, have to be replaced by their enclosing cluster. Implementing collapsing in this way, allows us to reuse the original graph-based clustering algorithms without modification.

4.3 Negative Information

Dealing with negative information is needed in a user-guided mode. Negative information allows the user to state negative links between entities (classes), "before" applying techniques, whose results will respect this negative information, as such that the user does not get confronted with clusters containing these negative information again.

Even while it would be logical to pass negative information to the techniques, so that they can deal with it, as already stated by Koschke, it would be a better solution to handle this information outside the techniques. This allows us to keep the implementation of the techniques, without making them more complex. For this reason, the implementation of handling the different types of negative information stated bellow, pre- or post-process the input or output of the algorithms.

4.3.1 Temporary Negative Information

It is possible that a class-based program features relations between classes which are only present for object-oriented purposes, as stated in section 3.2.4. As was explained there, it is useful for the user to be able to remove certain relations between classes. Removing such direct relations however, does not imply that it is not possible that clustering techniques cluster the elements together, caused by transitive relations. In other words, this kind of information is *temporary negative information*. In order to deal with this type of negative information, the input of clustering techniques is changed as such that the *temporary negated links* are removed.

4.3.2 Final Negative Information

Another type of negative information, is *final negative information*. This information is represented by mutually exclusive links between classes. This means that all results of techniques which actually place mutually exclusive classes together, have to be reconsidered. Koschke described a technique for splitting clusters containing mutually exclusive links, as such that the splitted clusters have a maximal amount of relations to between entities in the subcluster. This algorithm can be adapted straightforwardly as such that it is based on relations between classes.

4.3.3 Accepted Negative Information

When we are constructing software components, at certain points in time, certain components will be fully self containing. These components are software components which can be accepted by the user as *finished*. At that point, information about the rest of the subject system in relation to those components is irrelevant, and will only cause noise in the results of the techniques applied to the system. For this reason, by accepting components, all classes in the component get removed from the analysis. This stems from the idea that it is easier to find components in smal systems than in big systems.

4.4 Cluster Completeness

Looking for software components in a class-based setting, informally comes down to finding parts of class hierarchies which represent meaningful parts of the subject application which can be extracted from the subject application and can be reused later on in other applications. Once techniques find clusters of classes, this (informal) definition can be leveraged to complete candidate clusters into actual software components.

Most of the adapted automatic techniques presented in chapter 3 already automatically incorporate hierarchical information by using the *referred-by* relation which includes superclass references. However, especially after combining and overlapping different clustering techniques, it is possible that parts of hierarchies are missing. This results in extra tools for analysing candidate clusters which should be available as additional tools to the techniques.

The tools presented in the next sections are defined as additional tools to the clustering techniques, and should probably only be used right before accepting the cluster as a software component.

4.4.1 Cluster Requirements

When we find a cluster of classes, it is possible that the cluster refers to external classes. This does not necessarily mean that those external classes should be part of the component that the cluster represents, but the user should be able to analyse clusters for these types of classes easily, since they represent the *requirements* of the cluster. Since it is possible that requirement classes are classes which were missed by the used techniques, the user should be able to add such requirements with just a few mouse clicks. These kind of classes, because of the definition of *referred-by*, overlaps two types of classes:

- **Superclasses.** This is the set of classes of which the roots of the component its hierarchies subclass.
- **Referenced Classes.** This is the set of classes to which classes in the component directly refer.

For reusing software components with requirements, a new environment in which the software component is going to be used, should provide the same interface of classes listed in the requirements. When the requirements of the software component state library classes, this is probably not a problem. When classes are listed which are not part of the standard library however, before reusing the component, the engineer should make sure that the necessary classes are available. In other words, next to using such extraction to complete software components, it is also useful to automatically describe a crucial set of requirements.

4.4.2 Cluster Usage

In order to complete candidate clusters, it is useful to investigate how the environment from which we extract the clusters is using the cluster. This task is additional to calculating the requirements of a cluster. This is not only useful for completing clusters, but it also provides us with an automatic mechanism to describe what (probably[2]) presents the interface of the software component.

4.4.3 Inner Cluster Completeness

In the introduction of this section, we informally defined clusters as sets of parts of class hierarchies. These hierarchies do not need to be complete, since it is possible

[2]In case of non-abstract usage of the component in the subject system, the interface will be too big. Secondly, it is also possible that parts are to be defined as part of the interface, but which are not used by the environment. This occurs when components are designed more completely than they are actually used.

that clusters subclass from library classes. It is also possible that a part of the interface of a software component can be used as such, that classes subclass from certain interface classes. However, this does not say anything about classes of which a superclass is available in the cluster, as well as at least one subclass. When such internal classes are not member of the cluster, we call them **ghostclasses**. There is (probably) no reason why classes of this type are not present in the cluster, since the parts of the class hierarchies surrounding these classes, which *are* present in the cluster, are broken.

In order to reuse a cluster containing ghostclasses, we have to make sure that all ghostclasses are actually available in the system. As such, ghostclasses are part of the requirements of the cluster, although a more important kind, since they are more likely to actually be elements which belong to the cluster.

4.5 Summary

Since none of the techniques, presented in chapter 3, are completely accurate, in this chapter we presented ways of obtaining improved results. As Koschke did before us in his thesis, we presented ways of combining and manipulating input and results of the different techniques which are expected to improve results. Additionally, we showed how certain object-oriented information can be leveraged to complete candidate components, and to automatically extract requirements and candidate interfaces of software components.

5 The Adapted Interactive Method

In the previous chapter, we explained how component detection techniques can be combined in order to improve their results. We also showed how object-oriented information can be used to complete candidate components. However, it is clear that, due to the complexity and vagueness, component detection techniques can only go so far. It will probably remain a problem which has to be tackled in cooperation with a human engineer. As such, next to the research to component detection techniques, it is interesting to find ways for aiding the user so that he can wade through the information obtained by the techniques, as fast as possible.

In this chapter, we will present an interactive method for detecting software components. We will show how the user can conveniently be involved in the process of building components, and how the user can inspect components with as goal improving them.

5.1 Result Combination and Presentation

In order to involve the user in finding software components, we have built a user interface which allows the user to browse through results of all techniques at the same time. The basic results of clustering algorithms are called **reasons**, since they are mostly too basic to already be called components, but they are already reasons to cluster elements together. Each clustering technique can decide by itself how to present their results to the user. Mostly these ways are conform with the ways described in chapter 3. However, we have made a few adjustments to limit the amount of information presented to the user.

5.1.1 Global Object Reference

In section 3.2.1, we already stated that this technique suffers from more false positives due to the unification of global objects with interface classes. From case studies we have noticed that in most cases, when using this technique as an automatic technique, the biggest part of the subject application collapses into one big cluster. This is quite logical, since programs are mostly written as such that there is always a (transitive) connection between any two classes in a system, unless there are some classes which have a different type of entry point than a reference (for example, by using meta-information).

A second observation is that the algorithm which calculates these clusters, finds "global objects" and "procedures" which refer to them, and clusters them together into *basic reasons*. In the second part of the algorithm, all overlapping gets merged, until a *set of disjoint sets* remains.

Since we define the *space of components* in the same way as the second part of the algorithm, we only present the basic reasons to the user. These basic reasons are structured as such that they contain the referred class as well as all classes which refer to that class. The label of the reason is the referred class. In this way the user can accept labels to be *internal classes* as opposed to rejecting labels which are *interface classes*. By accepting and adding the correct reasons to the space of components, the overlapping reasons get automatically combined into new, bigger clusters.

5.1.2 Same Module Heuristic

The original *Same Module* heuristic as presented in the thesis of Koschke, actually applies the *Global Object Reference* heuristic, limited to the scope of modules. This equals the refining combination of a "heuristic" which would cluster all elements by module, with the *Global Object Reference* heuristic.

Since combinations of other heuristics with full modules can also be interesting to guide the user in finding software components, we present the *Same Module* heuristic in a different way from the original one. We fully show modules (which were in our case *packages*, *namespaces* and *class hierarchies*) to the user. If the user now wants to inspect what the results of the original heuristic would be, he just has to combine the new version of the results with the *Global Object Reference* heuristic.

An extra advantage of presenting the results like this will be that a user can cluster the program by its modules, and then inspect the resulting clusters using the available tools. In this way, the user can use the tool to inspect how well the original program is decomposed in *packages*, *namespaces* or *class hierarchies*.

5.1.3 Overlapping Results

Many of the techniques, such as all the *Same Module* and the *Delta-IC* heuristics, result in reasons which are possibly overlapping. In some cases, some reasons are subsets of other reasons. In order to limit the amount of information which the user has to observe, such reasons are incrementally (hierarchically) ordered before showing them to the user. This method complies with an extension to *Delta-IC*, presented by Koschke, where largely overlapping results are merged. However, by only merging fully overlapping reasons, we do not need an extra parameter (the amount of overlap minimally required), nor do we introduce more uncertainty, such that this technique can be reused for other techniques too.

We do not necessarily place reasons incrementally based on which elements are finally clustered by the techniques, but by the elements which caused the techniques to cluster the elements together. In order fulfill this task, techniques whose results are to be ordered incrementally, have to annotate reasons with the directed graph which lies at the origin of the clustering. In general this comes down to adding a set of *directed edges* to the reason. In order to place reasons incrementally, we then look for subsets of the end-points of edges.

$$\text{ends}(C) = \bigcup_{\text{edge}\in\text{directed-edges}(C)} \text{end}(\text{edge}) \tag{5.1}$$

$$\text{ifSubset}(S_1, S_2) = \begin{cases} \{S_1\} & \text{if } \forall e \in S_1 : e \in S_2, \\ \{\} & \text{else.} \end{cases} \tag{5.2}$$

$$\text{find-subclusters}(C, Cs) = \bigcup_{Cluster\in Cs} \text{ifSubset}(\text{ends}(Cluster), \text{ends}(C)) \tag{5.3}$$

When we place reasons incrementally in this way, since for example for *Delta-IC*, the less elements are referenced, the smaller the chance that elements only refer to that specific subset of elements, and thus the more significant but smaller the related reason is. As such this type of trees resemble semantically to the trees resulted from hierarchical clustering algorithms (such as the *Similarity Clustering* heuristic).

5.1.4 Dominance Analysis

Dominating Trees transitively dominate elements of its subnodes. As such, at each node of the tree, we can recursively combine the elements only dominated by the node, with elements dominated by its subnodes. If we present reasons in this way, and define *Dominance Analysis* as an *Incremental* heuristic, results of *Dominance Analysis* automatically gets ordered in dominating trees.

5.1.5 Crafting Reasons

Up until now we have mostly defined how different types of techniques can be combined or presented as such that they make good building blocks for clusters. The next step in the process is defining how a user can select such building blocks, and actually combine them into clusters. One way of doing this is already presented by the definition of the *space of components*, which covers combining reasons in an expanding way. The second way of combining reasons was refining.

We present reasons to a user such that he can select a reason as being the *current reason*. The default current reason is the *global reason* which clusters all classes of the subject system. When a user adds a reason to the current reason, the current reason gets refined by the new reason, so that the intersection of both reasons becomes the new current reason. A user can also customize the current reason, by manually selecting a subgroup of the current reason. At any point, a user can accept the current reason, which will then be added to the *space of components*. Refining reasons in this way provides the user with the power and insight of what is happening, at all moments.

A disadvantage of presenting results refinements in this way, is that a lot of the work has to be done manually, in other words, slowly. For this reason two other operators are provided which use the current reason, and which are modelled on the two different ways of refining results.

The first one enables the user to automatically add results of techniques to the current reason, and accept the combination. In this way, the *Same Module* heuristic by Koschke can easily be mimicked by selecting module by module as current reason, and *auto-merging* all (related) *Global Object References*.

The second one is built on the idea that a second technique should only be presented with the scope of the subgroup of elements, selected by the first technique. The result of the first technique is represented as the current reason. The second operator allows the user to find subclusters inside the current reason, by applying the techniques again to the current cluster. The user can build clusters using the results of the techniques applied to the current cluster. When all subclusters are found, the user can accept these subclusters.

5.1.6 Annotating Combinations

Original reasons are self explanatory, since they are linked to a clustering technique which constructed them. When we combine reasons into combined reasons (or clusters) however, it can be difficult to grasp which reasons lie at the basis of grouping exactly those elements together. For this reason, combined reasons (and clusters) are annotated with the set of all reasons (positive or negative) which added to the construction of that exact reason (cluster). By doing this, users can easily inspect how clusters and reasons were constructed, or decompose them into their original building blocks and refine them.

5.1.7 Filtering Results

In order to guide a user better and as fast as possible through the amount of reasons presented it is useful to have a good, extendible, filtering mechanism.

Elements which are already clustered by the user, in other words, which are already confirmed by the user as belonging together, should not be reconfirmed by the user. For this reason, all basic reasons which are subsets of already existing clusters in the *space of components* should be removed from the lists of candidates.

While refining a certain reason, by combining results of certain techniques, it is vital that only results related to the combined reason which the user is currently constructing, are shown. Since we refine reasons only until there are minimally two elements in the reasons[1], at any moment only reasons which have at least two elements in common with the reason we are currently constructing, are shown.

Another remark from a refinement point of view, is that reasons only refine a given reason, when intersection of both reasons is smaller than the original given reason. Results which do not actually refine the reason which we are currently crafting, are filtered too.

Filtering as defined above, improves the way the *Same Module* definition by Koschke can be mimicked. When we select a module, automatically only reasons which reference objects in the module are listed as results of the *Global Object References* heuristic. This way a user is only presented with relevant information and will have less trouble finding the internal classes related to the module.

5.1.8 Hierarchical Filtering

When a reason is structured hierarchically, and when a certain node is filtered away, it is possible that unlike the supernode, the subnode complies with the filter.

[1]The minimal amount of elements in a reason is two, since a reason is a basic building block for clusters. Elements are only clustered together if there are at least two elements.

Therefore, for all filtered supernodes, the filter has to be applied recursively to its subnodes. When a subnode does comply with the filter, it gets rebound to the first supernode which was not filtered away. If there is no supernode which was not filtered, the subnode gets added to the list of results.

If we now select a supernode of a hierarchical reason, as current reason, this type of filtering results (possibly only) in direct subnodes of the current reason as refining results. This presents an advantage for all heuristics where the user wants the find the correct upper and lower bound in a hierarchy, such as *Similarity Clustering* and *Dominance Analysis*.

For example, a user can select a node of a *Dominance Tree* as current reason, the point where we enter a cluster. By doing this, only subnodes will appear in the result of the *Dominance Analysis*. The user can now browse through those results to find subnodes which do not belong to the cluster anymore, but to libraries which were solely used by the cluster, and reject them from the current reason.

5.2 Component Evaluation

Once we have found a set of disjoint components, it is important that the user is presented with a set of tools to evaluate and complete these components, since the task of evaluating candidate components is comparatively large to the time needed to group candidates [Kos02].

Until now we already explained how components can be evaluated by ways of combining techniques in section 4.1, by investigating the requirements in section 4.4.1 and investigating the component usage in section 4.4.2. Moreover we annotated components with all reasons out of which the component is composed so that the user is able to browse the component to see its structure. We explained this in section 5.1.6.

Apart from making components browsable as hierarchical sets of elements, it is also useful to visualize them, in order to highlight certain relations inside the cluster, or between the cluster and its surroundings. The concept of visualizations is well-known as “the power or process of forming a mental image of vision of something not actually present to the sight” [SW89].

5.2.1 Visualization Type

In [MS00], L. Martin showed that the optimal way of showing dependencies between components, is by connecting the components by use of graphs. Class hierarchies are logically also represented by directed acyclic graphs. For this reason, we will represent most of our visualizations, which can be combinations of components and (parts of) class hierarchies, by graphs.

For the colours used in our visualizations, we follow visual guidelines by Bertin [Ber83], who states that the human mind can only process about a dozen different colours.

As stated earlier, there are two different aspects in visualizing components. First we want to visualize how components are constructed as a combination of reasons. Secondly, we want to investigate how components react to their environment.

5.2.2 Visualizing Component Composition

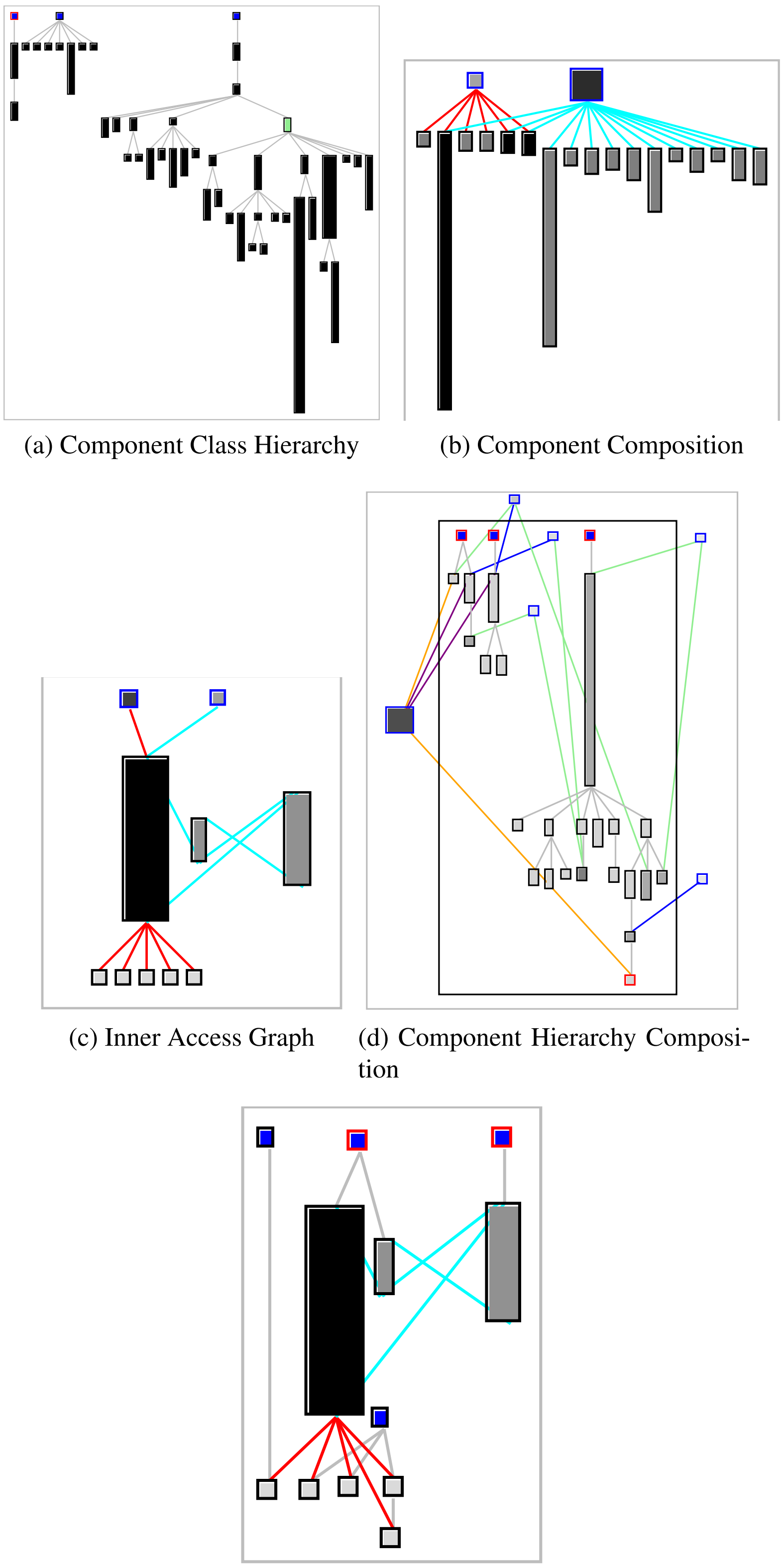
There are several reasons for visualizing the composition of components. Some of the reasons can be combined in one and the same visualization, but in order to maintain simplicity, sometimes it is more useful to split visualizations as such that they focus on specific problems. This results in three main different visualizations, of which the first one is combined with the two other ones into two extra combined visualizations. The overlay of visualizations allows us to conveniently present extra information. In some cases it will also allow us to present information in the other visualizations, in a cleaner and less obfuscated way.

Component Class Hierarchy

Visualizing the class hierarchy of a component lies quite close to the idea of cluster completeness presented in section 4.4. More specifically, when visualizing class hierarchies of components, it is interesting to highlight inner completeness (section 4.4.3) and show the superclasses of the component (section 4.4.1).

Applying *Component Class Hierarchy* visualizations might expose the presence of *ghostclasses*. Since inner completeness is one of the most important defects of class-based components, we colour these type of classes bright green so that they stand out. The superclasses of the component are classes which are needed by the component, but which are not actually available in the component. In order to highlight this, they also get a separate color, blue in our case. For the rest of the classes in the hierarchy, we apply a standard class hierarchy visualization scheme in which class-metrics are used for the outlook of the rectangle which represent them.

In this way, users can, by one look see which (parts of) class hierarchies are present in software components. They can inspect which superclasses are needed so that the component can be installed, as well as which classes are missing from it.



(a) Component Class Hierarchy

(b) Component Composition

(c) Inner Access Graph

(d) Component Hierarchy Composition

(e) Inner Hierarchy Access Graph

Figure 5.1: Different Component Composition Visualizations

Component Composition

If components are composed of multiple reasons, possibly resulting from different clustering techniques, it is not always clear which reasons agree with the presence of which classes in a cluster. For this reason a visualization which highlights agreement of reasons on the composition of a component, is useful.

In order to visualize this scheme, which is actually an overlap of sets, there are multiple possibilities. In mathematics, it is a standard procedure to visualize overlapping sets using Venn diagrams [Wei99]. However, this technique quickly gets out of hand, since for n reasons, there are $\sum_{i=0}^{n}\binom{n}{i}$ or 2^n different overlapping subsets to be displayed. In order to keep things more simple, we also use a directed graph.

The edges of the graph go out from icons (squares with blue borders) which represent the reasons, to all elements represented by the reason. Every reason is assigned its own colour of edges, selected from a limited selection of colors. For the colour of the squares representing reasons, we use a metric which measures how many of the classes in the component are supported by the reason. The size of the square indicates how many elements are truly supported by the reason. The amount of supported elements can be bigger than the amount of represented elements, since a part of the reason can be cut away by a refinement. The colour we use for squares representing classes, is a metric indicating how many of the total amount of reasons of the component support its presence.

Inner Access Graph

Some of the techniques presented in chapter 3, are centered around access graphs. The most important ones here are, *Dominance Analysis* and *Strongly Connected Components*, but also *Global Object Reference* to some extent belongs to this category. Since this information is used as basis for clustering classes together, it might be interesting to save this access information in reasons, so that combinations of access graphs of the different reasons can be shown later. In section 5.1.3 we actually already annotated reasons with access information, so here that information is reused.

The most interesting access graphs that will be visualized in this way, are the dominating trees resulting from dominance analysis. Since they are trees, it is interesting to order them as such.

Different reasons can be incorporated in this visualization in a similar way as for the previous visualization, however here we can leverage structural information in order to limit the amount of edges between elements, Which improves the understandability of the total visualization. We do this, by only drawing edges from reasons to the different entry points of access graphs. For each dominating tree, as well as for each strongly connected cluster (which are cyclic graphs), there is only one such entry point. In order to visualize which of the classes of the tree or graph are actually supported by the reason, the access edges between elements are coloured in the same way.

Access graphs are an ideal way to detect *extended data classes*, defined in section 3.2.4. Classes which are linked by an unusual amount of edges are most likely to be such kind of classes. Therefore, in order to detect those classes easily, we colour classes related to the amount of edges.

Component Hierarchy Composition

The *Component Composition* visualization shows interesting information about general sets of entities. However, since the entities we are clustering are ordered by nature in class hierarchies, it is interesting to overlay both the *Component Class Hierarchy* and the *Component Composition* visualization.

We can use the hierarchical information of *Component Class Hierarchy*, to limit the amount of edges going out from reasons to classes, in a way similar to *Inner Access Graph* edges of section 5.2.2. However, here we connect included classes with a green edge, and fully included subhierarchies with a blue one. It is also possible to invert edges from a reason. If there is only a limited amount of classes *not* included in a reason, an orange edge means that a class is not included and a purple that a subhierarchy is not included. Orange or purple edges coming from a reason, means that everything which is not connected to the reason, is automatically supported. The visualization automatically choses the version which produces the least amount of edges.

This visualization might expose interesting information about how parts of hierarchies are joined by different reasons. Still, the original version is also still interesting as separate visualization since class hierarchies might also obscure the actual component composition.

Inner Hierarchy Access Graph

In the same line as the overlay of *Component Composition* and *Component Class Hierarchy* in section 5.2.2, we can also overlay the *Inner Access Graph* of section 5.2.2 with the *Component Class Hierarchy*. This technique highlights how classes in a component, while ordered as class hierarchies, access each other. In order to limit the amount of edges, we do not show *referred-by* edges resulting from hierarchical information, since by definition the set of *referred-by* relations includes links from classes to its superclasses.

5.2.3 Visualizing Component Interaction

At the moment we visualize the interaction of a component with its enclosing environment, we expect those components to already comply with the inner completeness restriction. Because of this restriction, parts of class hierarchies are fully enclosed in such components, which makes it easy to represent such components as squares containing (parts of) the class hierarchies.

There are two different sources of information which can be used to visualize interaction between components and its environment. It is possible to show how components refer to classes (or components) in their environment. Secondly it is possible to show the class (and component) hierarchies which are used as superclasses and which classes and components subclass from the visualized components. Because the structure of accesses outside and between components has no importance, we only present the access visualization, as a visualization projected on the class hierarchy visualization.

In order to limit the amount of information shown, we only show classes hierarchies which are related to the visualized components.

5.2.4 Visualizing Reasons

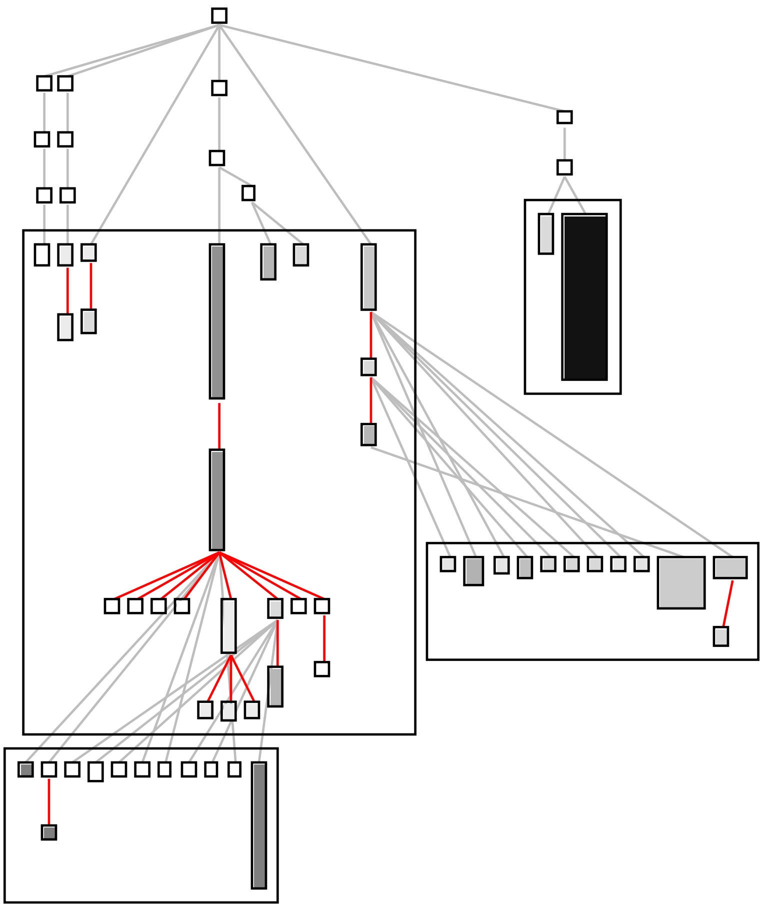
Next to the fact that reasons can be combined into components, reasons can also be combined into reasons. Components are actually hierarchical combinations of reasons. When we visualize a component based on combined reasons, the difference between different subreasons automatically gets hidden, by interpreting the combined reason as one. Now because of this idea, it is interesting to also be able to visualize composition of reasons in a similar way to the composition of components.

5.2.5 Fine-Grained Visualizations

In some situations, it might be necessary to take a closer look at what exactly constituted to the clustering of certain classes. For this reason, it is useful to be able to display a group of *class blueprints* [Duc05] of the selected classes. In this way, a user has more insight in the actual structure and dependencies of the cluster.

5.3 Summary

In this chapter we described a way to optimally present results of component recovery techniques to a user, to aid him to evaluate those results in order to find fully self containing software components. We reevaluated how to present the results of the adapted component detection techniques defined in chapter 3. Then we presented ways to combine results into components based on combinations presented in chapter 4. Finally, we discussed new ways of evaluating components by use of component visualizations.


(a) Cluster Hierarchy

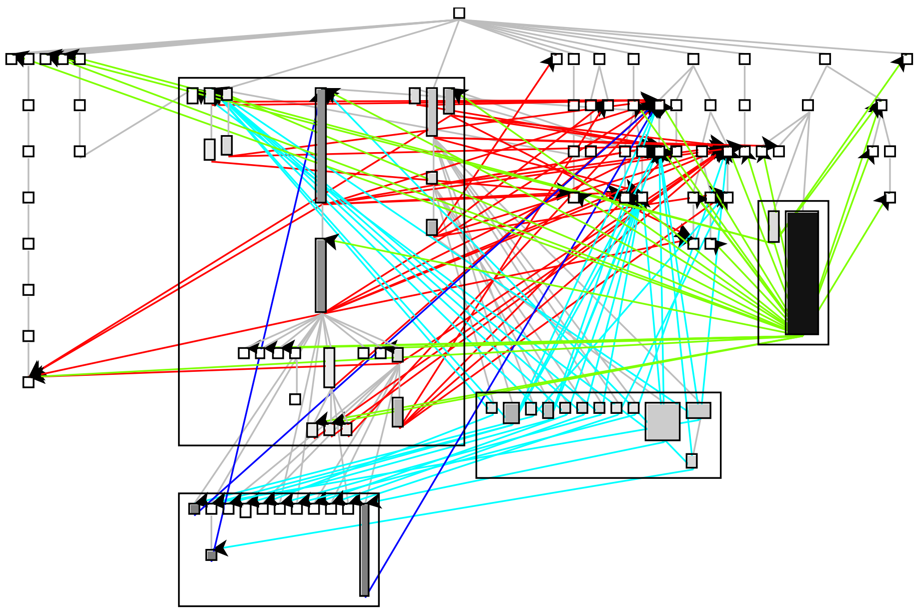

(b) Accesses Between Hierarchical Clusters

Figure 5.2: Component Interaction

6
Case Study

In previous chapters, we have explained how the atomic component detection techniques, described in the thesis of R. Koschke [Kos02], are technically adapted to be applied in a dynamically-typed object-oriented environment, using only explicit dependencies. We have showed that object-oriented properties are leveraged to inspect and complete the results from these techniques. In this chapter we complement the theoretical descriptions with case studies validating their use.

6.1 Prototype Implementation

In order to perform case studies, we have build a prototype tool called *ClusterFinder* which incorporates the techniques described in chapters 3, 4 and 5. It is implemented in Cincom Smalltalk [cin], as a tool extension of the language independent reengineering environment Moose [NDG05]. Our tool uses explicit dependencies in the object-oriented language meta-model *FAMIX* (FAMOOS Information Exchange Model) [DTD01] as basis for adapted techniques. Visualizations are constructed using the agile visualization framework *Mondrian* [MGL06]. The interface of the prototype is described in more detail in Appendix A.

6.1.1 Limitations

At the time of the experiment, the *ClusterFinder* tool had some limitations.

- As a limitation to negative information, it was only possible to (temporarily of fully) remove classes. Thus, it was not possible to specifically remove certain relations between classes, or to introduce *mutually exclusive* relations. This has an impact on the studies done around *extended data classes*, where negative information is crucial. This results in the fact that, wherever relations have to be negated, one of the related classes has to be fully ignored.

- Even while the clustering techniques plugins are implemented using different parameters as described in chapter 3, no interface is provided for easy adjustment. So in this chapter, we do not investigate how techniques can be adjusted by playing with different parameter settings. However, for techniques such as *Delta-IC*, where a minimal Θ should be set, this is no problem since we just display all results in an ordered way as such that the user can choose to ignore results results below a minimal Θ.

- To display visualizations, we use the framework Mondrian [MGL06]. This framework has a default set of layouts which can be used to order elements, such as a *tree layout*. However, since the *Interaction Visualizations* which we described in section 5.2.3 uses hierarchical trees (trees of which the nodes can contain up to forests of subtrees), a normal tree layout is not sufficient. A better layout for such trees was not provided, resulting in visualizations which are not optimally ordered.

- Interface extraction was not available.

6.2 Case Study: *Moose Namespace*

We validate our techniques by applying them to the reengineering environment *Moose*, in which we embedded our tool.

6.2.1 Statistics

When we “moosify” the subject system, in other words, import it into the moose system as such that a meta-model representation in FAMIX is available, moose provides the information listed in table 6.1. The used terminology is as follows:

Model vs Stub Classes In these statistics, a difference is made between *model classes* and *stub classes*. This difference is adopted directly from Moose, which tags imported classes as *non-stub*. Classes which are only imported because they linked to the model classes in some way, but not part of the

Model Classes	*Stub Classes*	$\sum$ *Classes*	*SLOC*
339	323	662	20.869

Table 6.1: Statistics

classes which were selected, are imported as *stub classes*. When a class is tagged as a stub, this means that only a minimal amount of information is available about those classes. When detecting components in this environment, it makes sense to incorporate the information about links between model classes, and stub classes to some extent, but it probably does not make sense to look for clusters in the set of stub classes. For this reason, our tool provides a “clustering algorithm” which clusters classes by their state of the *stub* attribute. By selecting the cluster for which that attribute is *false* as current cluster, we automatically only look for clusters in the set of model classes without excluding information about the referred stub classes.

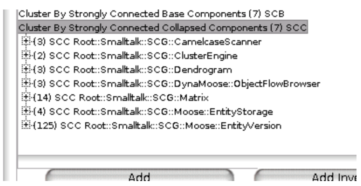
In order to visually support the difference between model and stub classes, model classes have a black border, and stub classes a red border around the rectangles representing them.

SLOC is the total amount of lines of code of the system S. This is calculated as $\sum_{c \in \text{classes}(S)} LOC(c)$. Remark that stub classes are defined to have 0 lines of code, thus including them in the definition of $SLOC$ has no impact.

6.2.2 Candidate Component Detection

We start our experiment by applying the *Strongly Connected Components* algorithm to the SCG Namespace. We use this algorithm first, since strongly connected components imply that the classes in the connected components need each other. This was also specified as first step of the *component detection strategy* in [Kos02].

The resulting clusters have sizes 2, 3, 3, 3, 4, 14 and 125 (figure 6.1). Since the last one consists of 37% of the model classes, which is an unusually big partition of the system, we investigate this cluster further before accepting it. If we apply the *inner hierarchy access graph* visualization of section 5.2.2, this results in figure 6.2



```
Cluster By Strongly Connected Base Components (7) SCB
Cluster By Strongly Connected Collapsed Components (7) SCC
  (3) SCC Root::Smalltalk::SCG::CamelcaseScanner
  (2) SCC Root::Smalltalk::SCG::ClusterEngine
  (3) SCC Root::Smalltalk::SCG::Dendrogram
  (3) SCC Root::Smalltalk::SCG::DynaMoose::ObjectFlowBrowser
  (14) SCC Root::Smalltalk::SCG::Matrix
  (4) SCC Root::Smalltalk::SCG::Moose::EntityStorage
  (125) SCC Root::Smalltalk::SCG::Moose::EntityVersion
```

Figure 6.1: 7 Clusters

If we look closely, we see that there are only few classes with a big amount of access relations,

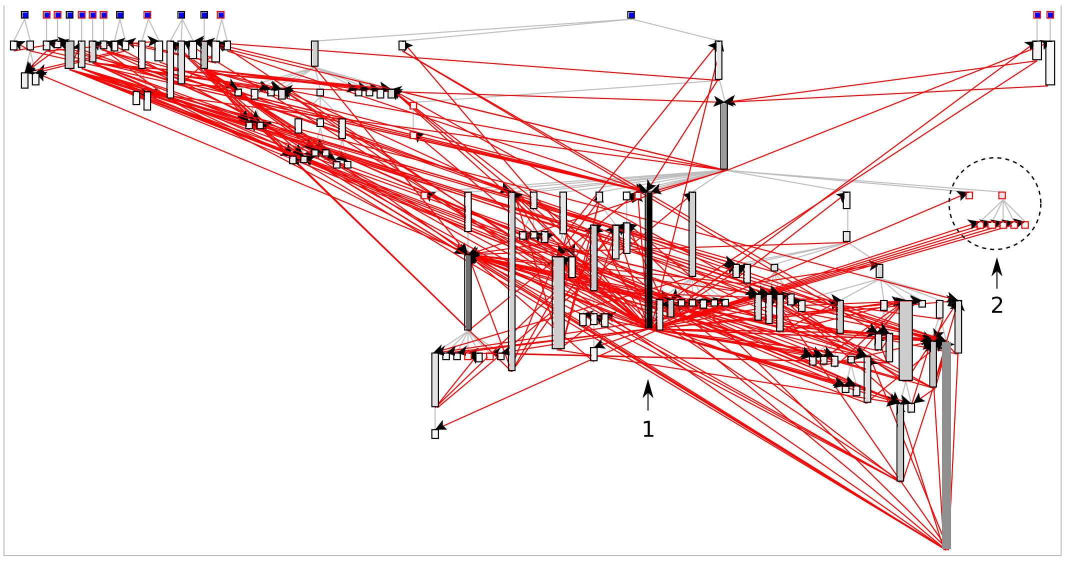


Figure 6.2: An Oversized Strongly Connected Component

as opposed to multiple classes with only few access relations. This indicates that probably these few classes hold together multiple small strongly connected components, and as such are possible *extended data classes*.

Secondly, on the right of the image, we see a small subhierarchy of stub classes which are strongly connected, because of its hierarchy construction, to the class with most accesses in the cluster. This group is indicated with a circle and tagged 2. Since they are stub classes, they are not supposed to be part of the system, but only used by the system. If classes of this type are part of a strongly connected component, this clearly indicates that the relations pointing to those stub classes are only present to serve the implementation of the system partly composed of those stub classes. Therefore, the class from which the relations go out, is an *extended data class*.

Refinement

In order to detect if there might be useful strongly connected subcomponents in the oversized component, we remove *extended data classes* one by one, starting at the biggest one. The first class we remove is *SCG::Moose::MooseModel*, which is represented by the darkest rectangle, tagged 1, in figure 6.2. This results in four remaining subcomponents, presented in figure 6.3. Clusters labelled 3 and 4 are accepted, but bigger clusters 1 and 2 need some more inspection.

Cluster 1 appears to contain another *extended data class*, *SCG::Moose::Group*, which has a relation to a stub class. We refine this cluster by looking for *Strongly*

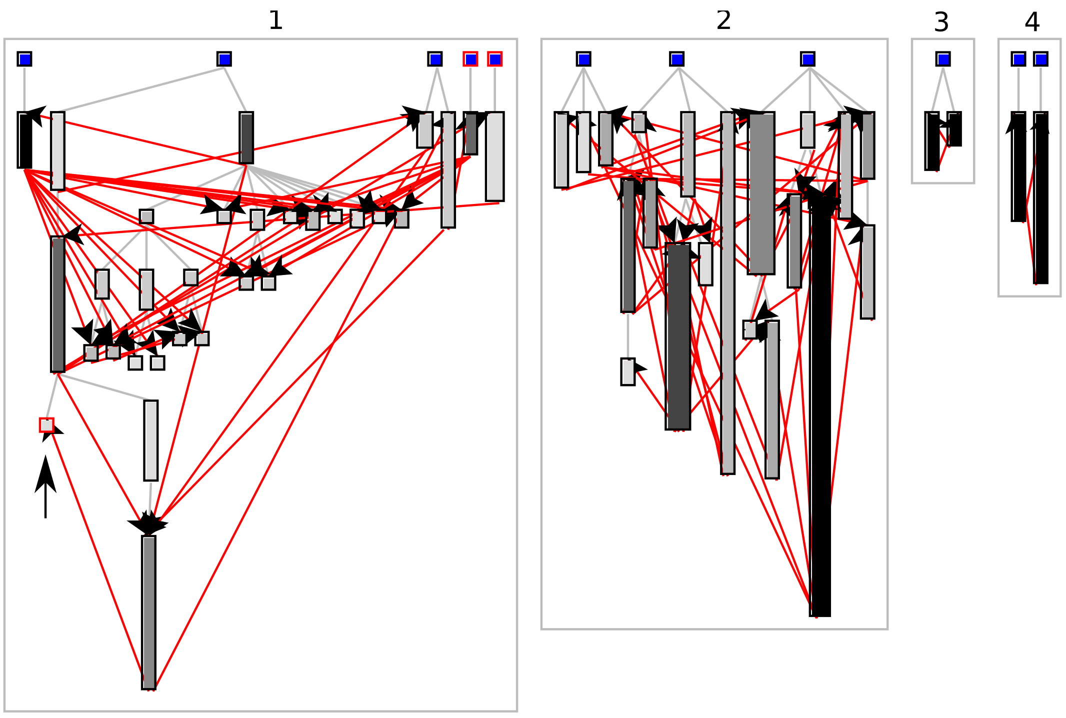


Figure 6.3: Reduced Strongly Connected Components

Connected Components inside the cluster, ignoring relations to the *extended data class*. This results in two clusters with sizes 2 and 21, each containing a class implementing a user interface and *one* subhierarchy. We accept these subclusters.

In cluster 2, we note that three different subhierarchies are quite strongly connected. However, it is possible that not just one class of a hierarchy is used as a *extended data class*, but that one of the hierarchies is designed as a *extended data hierarchy*. In order to test the level of relatedness between the hierarchies, we refine the cluster using *Similarity Clustering*. This rates the total similarity of the cluster at 7%. The first subclusters both have a similarity level of 24%. This places one of the three subhierarchies in the original cluster into a separate cluster. We accept these two clusters of sizes 8 and 12.

So far we have split the original cluster of 125 elements, into 6 clusters of sizes 2, 2, 2, 8, 12 and 21. These clusters represent *crystallization points* [Kos02].

Extending Results

We build further on top of these clusters by combining previous results with *Delta-IC* candidates with a high $\Delta IC(C)$, and with *Similarity Clustering* results with high similarity, which receive visual confirmation[1]. Here we must remark that these techniques sometimes cluster classes which perform a similar task, and as such are probably even subclassed from a same set of superclasses, but which actually belong in different environments. This is especially true for user interface classes, which generally use several of the same user interface elements, but which belong to the separate applications or tools for which they are defined. However, the fact that classes of this type are grouped together by some techniques, also indicates that they are probably mere *add-ons* to the original applications for which they were designed, and that the *back-end* of the original applications can be reused without it.

For this reason, to improve the global effectiveness of automatically applying these techniques, it is useful to ignore these *add-on clusters* before merging their results with results of other techniques. They can easily be detected by visualizing the results of these techniques using the *Component Interaction* visualization. These type of clusters generally have a bundle of access relations coming from all classes in the cluster, to a same set of classes, and separate bundles of access relations from classes in the cluster to different parts of the subject system.

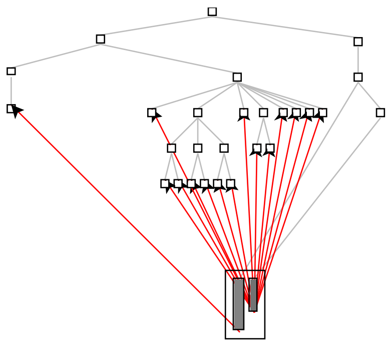


Figure 6.4: Add-on Cluster

If subclusters can be found inside *add-on clusters* of which both types of relation-bundles point to the same sets of classes, it is useful to accept these subclusters since they probably collaborate in their similar add-on task. This can be checked by using the *inner access graph* visualization.

Component Evaluation

Figure 6.5 shows the *component interaction* visualization of the components found in the breakup of the strongly connected cluster of 125 classes (visualized in figure 6.2), using the *Strongly Connected Components*, *Delta-IC* and *Similarity Clustering* techniques. Below we present a description of the different clusters, numbered as in the figure. Remember that this figure only represents 37% of the full application that we are studying.

1. Represents the core of the *Moose* application which was present in the

[1]Our user interface presents classes in the form *namespace::subnamespace::classname*, which already helps the user into deciding whether or not clusters are consistent. If two classes of unrelated namespaces are joined, this is doubtfully a clean join. Of course, this is only useful for systems which use different namespaces and/or naming schemes which have semantical meaning.

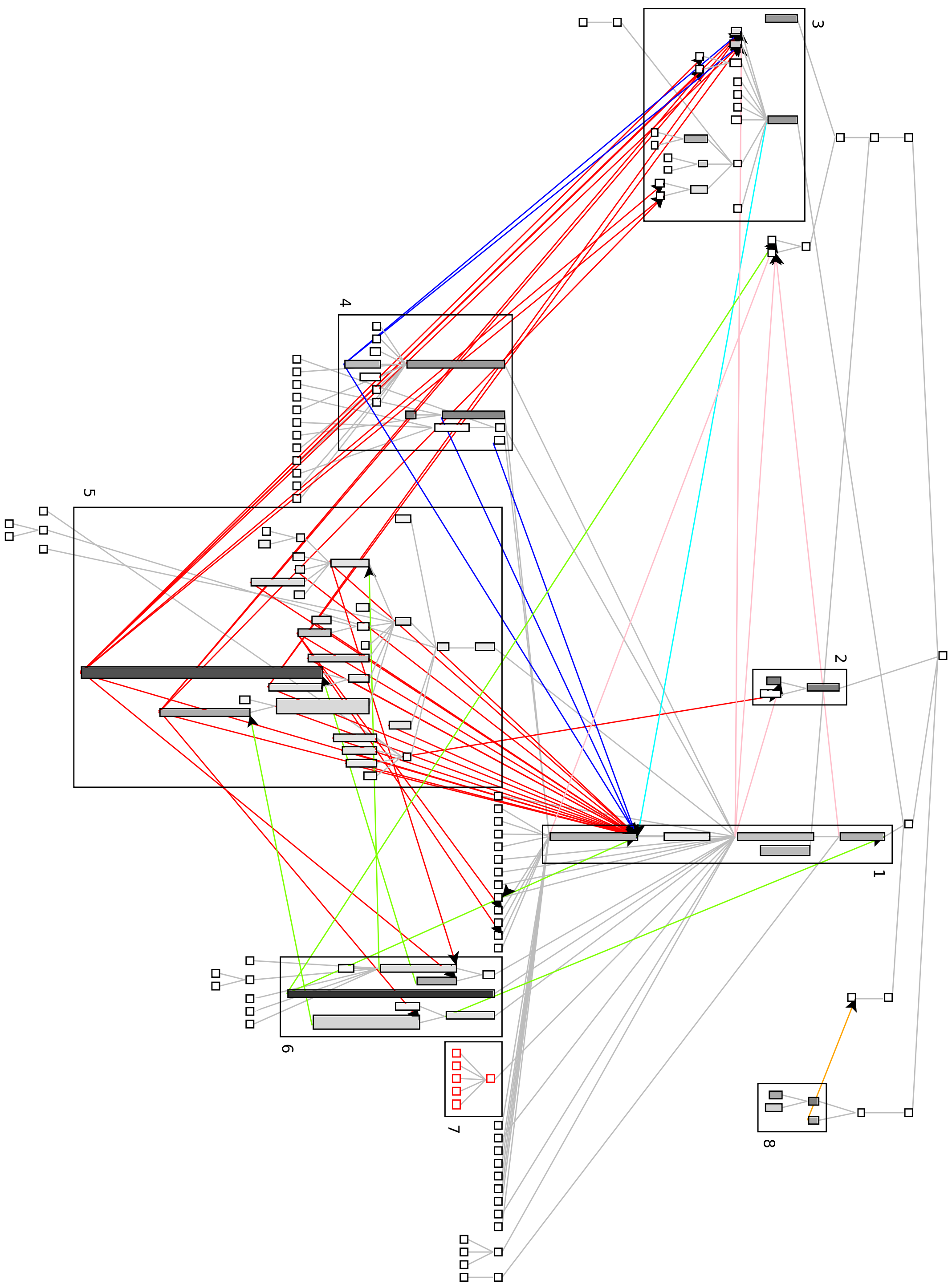


Figure 6.5: Interaction Visualization of Candidate Components

strongly connected cluster. All parts subclassing from the cluster, are extensions specifically designed for *Moose*.

2. A small library hierarchy used by component 5. The library has a reference to a class in the core of *Moose*, namely *SCG::Moose::Group*, which shows that the library was built with *Moose* in mind.

3. A bigger library hierarchy, extended with a user interface class, used by components 4 and 5.

4. This cluster represents a tool built on top of *Moose*. Therefore it uses its core. It also uses the library cluster 3.

5. Cluster 5 could also be considered a tool built on top of *Moose*. However, this cluster relies heavier on its core, as well as the library cluster 3 and more interestingly, another tool cluster, number 6. This indicates that one of both clusters contains a hierarchy of *extended data classes*. Since cluster 5 also references to other classes (which are not clustered yet), and cluster 6 only links to a user interface class next to the core cluster and cluster 5, we identify cluster 5 as possible *extended data cluster*.

6. This cluster is explained in the description of cluster 5.

7. Represents a part of a tool which was linked by an *extended data class*. Since it is a cluster of stub classes, it is semantically not really useful, and might as well be discarded.

8. Another library cluster. In this cluster we have not found yet for which other cluster it was designed.

Component Completion

The next step, after identifying (parts of) candidate components, is to complete them into full components. To perform this task, we start by looking at the different heuristics we have not used yet. We start by generating dominating trees of the *Dominance Analysis*, based on the clusters we have already found. In those results, we carefully select and/or reject candidate clusters which may or may not belong to same clusters

Then we use the *Global Object Reference* technique in combination with the *Interaction Visualization*, in order to detect which classes are not interface classes. This last step is quite *fuzzy* and therefore, this technique by itself is less useful than the others, if we do not have knowledge about the subject system, or if the subject system is badly designed. In our case however, we can heavily rely on correct usage of namespaces and naming schemes to validate candidate clusters[2].

After using *Global Object Reference*, we visualize the hierarchy of clusters, in order to detect if parts of class hierarchies are still unbound. If so, we complete those parts of the hierarchy as such that the unlinked parts are added to their parent cluster.

Finally we check the *requirements* of all clusters. All ghostclasses (in our case there are no ghostclasses detected) are added to their enclosing component. Requirements have to be checked manually to detect if they belong to the cluster or to a required cluster. The *interaction visualization* is useful to detect how such requirements are used in the rest of the application, which we need to know in order to make such a decision.

Accepting Results

After applying the last techniques, we accept the *intermediate* results we have found in the *oversized strongly connected cluster* as candidate clusters for the *Moose* application. Next, we evaluate the other *Strongly Connected Components* as stated in the beginning of section 6.2.2, and we loop the full process until the cluster we have refined at the moment we accept its results, was the cluster of all classes in the subject application.

6.3 Summary

In this chapter we validated our techniques by applying them using a prototype implementation called *ClusterFinder*. In doing so, we also implicitly described the following detection strategy:

1. Use *Strongly Connected Component* detection as starting point

2. Apply *Delta-IC*, *Dominance Analysis* and *Similarity Clustering* to extend the *Strongly Connected Components*, and possibly add new components.

3. Complete clusters using *Global Object Reference*, *Requirements Detection* and visualizations.

4. Whenever a detected cluster is identified as too big, we recursively apply this strategy to the oversized cluster, in order to refine it.

[2]If we were to rely too heavily on clustering by namespaces, this would result in a system "collapsed" in different namespace clusters.

7 Conclusions

As explained in Chapter 1, in order to *understand* the code of software systems, a *mental model* of the system has to be generated. Even while better encapsulation improves the understandability of systems, it is a difficult and time-consuming task in case of complex and/or badly designed systems. Since such a big share of time of maintaining applications is spent on understanding applications, the research to methodologies for automatically extracting conceptual models is justified.

7.1 Problem Statement Revisited

Research has already conducted to finding ways of automating the extraction of conceptual models. This research is mostly focussed on procedural programs. This is quite normal, since until a few years ago, this type of programs represented the biggest share of industrial applications. However, with the uprise of object-oriented programming, the need for automating the extraction of conceptual models from object-oriented programs has increased too.

The general ideas behind recovering conceptual models in procedural code are quite similar to those behind recovering conceptual models in object-oriented code. Unfortunately, the results from research conducted around procedural programs, are defined in terms of procedural programming language constructs, such as global objects and procedures. For this reason they can not be applied directly in object-oriented programs, because these constructs are not available. More correctly, these constructs are available in a different, encapsulated way.

7.2 Adapting Automatic Techniques

In this thesis we investigated how some of the procedural techniques are adapted so that they are applicable to dynamically-typed, object-oriented software. The techniques we chose as groundwork for our research, were the techniques described in the thesis of Rainer Koschke [Kos02]. This results in the adaptation of six techniques.

7.2.1 Shifting Focus

Migrating from the procedural to object-oriented programming paradigm results in several observations which affect the definition of most of the techniques. Due to this fact, we have specified general adaptations which are applied to most of the techniques.

- Instead of global objects and procedures, classes are the elements to be grouped together in components.
- Instead of procedure calls and global object references, techniques are modelled using class and superclass references.

7.2.2 Automatic Techniques

In addition to the more general adaptations, some of the techniques also require specific adaptations.

Same Module Heuristic is changed as such that it groups together classes based on the packages, namespaces or class hierarchies in which they were defined. This as opposed to grouping procedures and global objects in separate files together.

Similarity Clustering uses extra information about its elements to decide if they belong to the same component. We have replaced the definition of *informal information* so that it represents statically available information about classes. In addition to direct relations, *method names*, *attribute names*, *all superclasses*, *packages* and *namespaces* are considered.

7.2.3 Adaptation Problems

While adapting the automatic techniques, we identified several problems.

- By unifying procedures and global objects into one construct, it is possible that techniques based on references to global objects will be used in a setting were a class-reference actually represents a reference to a method.
- Excluding invocation candidates as information for the techniques results in more *true negatives*.
- As opposed to procedures, classes may contain a large amount of references.
- Certain object-oriented programming patterns decrease effectiveness. Here we identify *extended data classes* as a source of oversized connection- and graph-based clusters.

7.3 Combined and Incremental Techniques

Component detection techniques on their own can only go so far. To improve the results, as well as Koschke did [Kos02], we presented ways to combine different techniques. We identified two ways to combine results. Firstly by complementing each other and secondly by refining each other. While refining components, it is possible to include or ignore information about relations between the elements inside and the elements outside the component which we are refining.

7.3.1 Component Completion

We identified that hierarchical information can be leveraged to fill in "holes" found in object-oriented components. There are three main types of holes which can be detected

The requirements of a component are defined as the set of classes it refers to. This is the union of the set direct superclasses of the classes in the component and the set of referenced classes, minus the classes in the component.

The interface of a component is defined as the set of classes which refer to the component. This is the union of the set of direct subclasses of the classes in the component and the set of classes which reference the classes in the component, minus the classes in the component.

Inner Completeness is related to the completeness of subhierarchies found in a component. When a subclass of a class in a component is also found inside the component, all classes on the inheritance path from the subclass to the class also should be included in the component. If a class on the inheritance path is not included, this is called a *ghostclass*.

Requirements and interface classes are possible holes in components, while ghostclasses are definite holes.

7.4 Component Evaluation

Finally, we defined ways, additional to component completion, to evaluate candidate components based on component visualizations. These visualizations visualize the inner composition of a component, or the relationship between components and their environment. Three different visualizations are identified.

The Composition visualization is only defined for the inner composition of a component. This visualization shows how different building blocks are combined into the creation of the component.

Hierarchical visualizations highlight how components are related to class hierarchies. When we focus on the inner composition of a component, this visualization highlights ghostclasses.

When we focus on component interaction, this visualization shows how system hierarchies are divided over components.

Access visualizations show how classes and components access each other. When applying this visualization to the inner workings of a class, this might help the software engineer to detect *extended data classes*.

Applying it to a set of components allows users to detect certain relations between components. This is especially useful to detect *add-on components*. This is a type of component where classes are falsely grouped together because they possess similar characteristics. However, these characteristics only represent which value they add to the component to which they actually belong. User interface classes are typical victims of such clustering.

7.5 Future Work

In this section, we present a list of related topics which might be interesting for future research.

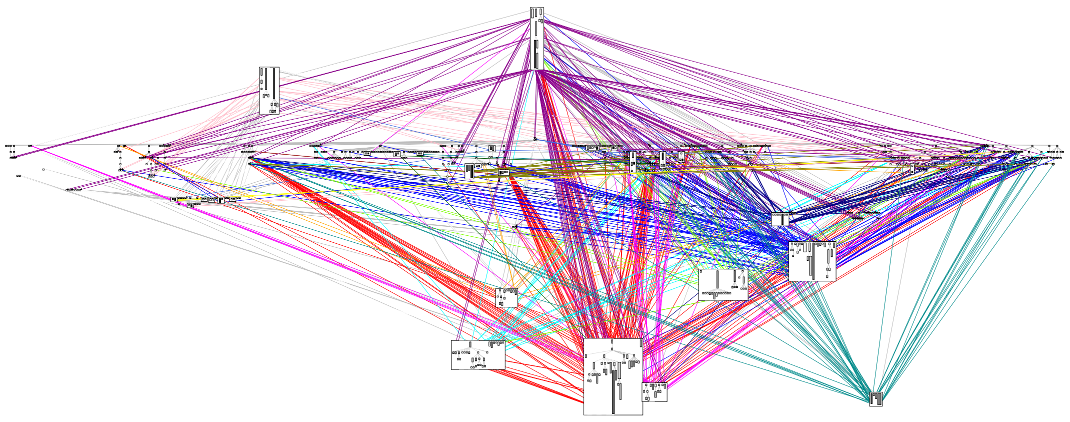


Figure 7.1: Overloaded Component Interaction Visualization

- The *Component Interaction* visualization, which was described in section 5.2.3, has scalability issues. When applying it to several *library components*, resulting visualizations are overloaded with information, as shown in figure 7.1. This might be solved by simplifying the image by presenting components as shown in figure 3.1, since in most cases several bundles of lines go out from components to only a few classes.
- The way of refining components presented in section 4.1.2, has as a disadvantage that it requires quite a lot of input of the user. It would be interesting to investigate how this can be improved, for example by using the *voting approach* described in [Kos02].
- A standard approach to validate the usefulness of new techniques, is to statically prove that they improve existing approaches. This still has to be done for our adapted and complementing techniques.
- We already defined certain ways to improve how a user can browse through results of the techniques. However, it is still open to investigation if there are other ways to improve the interaction between the techniques and the user. This is important since feedback of a maintainer will always be needed.

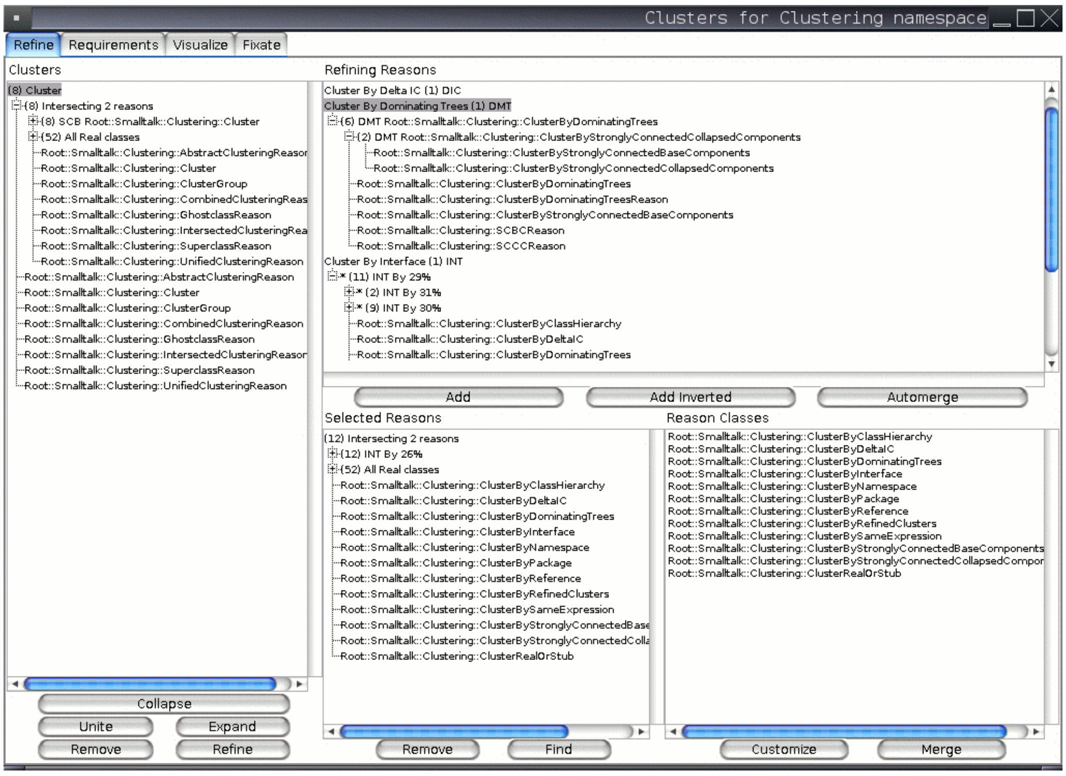
A
Prototype

This appendix describes the prototype of the adapted techniques described in this thesis.

Main Tab is shown in figure A.1a. The results of the clustering techniques are integrated in the hierarchical *listbox* labelled *Refining Reasons*. Listbox *Selected Reasons* represents the *current reason* defined in 5.1.5, and listbox *Reason Classes* shows the classes represented by the current reason. Listbox *Clusters* is the browsable list of accepted clusters in the *space of components*.

The *Add* button is used to refine the current reason. *Add Inverted* removes the classes in selected reason from the current reason. *Customize* allows the user to manually select and accept a subset of classes in the current reason. *Find* opens a new *ClusterFinder* which looks for clusters in the current reason. *Collapse* initiates the recalculation of all component detection techniques which are affected by collapsing clusters.

Requirements Tab in figure A.1b shows the same list of accepted clusters as in figure A.1a. By selecting a cluster and clicking *select*, the different types of requirements get calculated and presented in the three different lists. The requirements can be added to the cluster by selecting them and clicking the *merge* button under the list in which the requirements were selected.



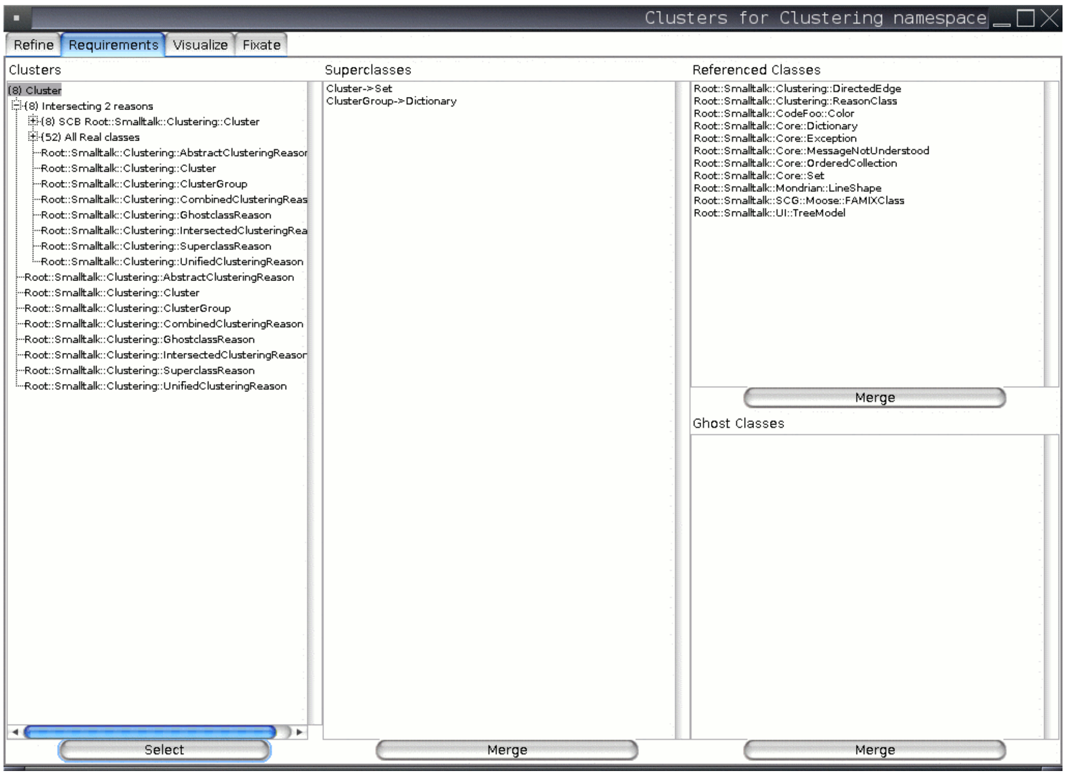
```
Clusters
(8) Cluster
(8) Intersecting 2 reasons
  (8) SCB Root::Smalltalk::Clustering::Cluster
  (52) All Real classes
   Root::Smalltalk::Clustering::AbstractClusteringReasor
   Root::Smalltalk::Clustering::Cluster
   Root::Smalltalk::Clustering::ClusterGroup
   Root::Smalltalk::Clustering::CombinedClusteringReas
   Root::Smalltalk::Clustering::GhostclassReason
   Root::Smalltalk::Clustering::IntersectedClusteringRea
   Root::Smalltalk::Clustering::SuperclassReason
   Root::Smalltalk::Clustering::UnifiedClusteringReason
 Root::Smalltalk::Clustering::AbstractClusteringReason
 Root::Smalltalk::Clustering::Cluster
 Root::Smalltalk::Clustering::ClusterGroup
 Root::Smalltalk::Clustering::CombinedClusteringReason
 Root::Smalltalk::Clustering::GhostclassReason
 Root::Smalltalk::Clustering::IntersectedClusteringReasor
 Root::Smalltalk::Clustering::SuperclassReason
 Root::Smalltalk::Clustering::UnifiedClusteringReason
```

```
Refining Reasons
Cluster By Delta IC (1) DIC
Cluster By Dominating Trees (1) DMT
 (6) DMT Root::Smalltalk::Clustering::ClusterByDominatingTrees
   (2) DMT Root::Smalltalk::Clustering::ClusterByStronglyConnectedCollapsedComponents
     Root::Smalltalk::Clustering::ClusterByStronglyConnectedBaseComponents
     Root::Smalltalk::Clustering::ClusterByStronglyConnectedCollapsedComponents
   Root::Smalltalk::Clustering::ClusterByDominatingTrees
   Root::Smalltalk::Clustering::ClusterByDominatingTreesReason
   Root::Smalltalk::Clustering::ClusterByStronglyConnectedBaseComponents
   Root::Smalltalk::Clustering::SCBCReason
   Root::Smalltalk::Clustering::SCCCReason
Cluster By Interface (1) INT
 * (11) INT By 29%
   * (2) INT By 31%
   * (9) INT By 30%
   Root::Smalltalk::Clustering::ClusterByClassHierarchy
   Root::Smalltalk::Clustering::ClusterByDeltaIC
   Root::Smalltalk::Clustering::ClusterByDominatingTrees
```

```
Selected Reasons
(12) Intersecting 2 reasons
 (12) INT By 26%
 (52) All Real classes
  Root::Smalltalk::Clustering::ClusterByClassHierarchy
  Root::Smalltalk::Clustering::ClusterByDeltaIC
  Root::Smalltalk::Clustering::ClusterByDominatingTrees
  Root::Smalltalk::Clustering::ClusterByInterface
  Root::Smalltalk::Clustering::ClusterByNamespace
  Root::Smalltalk::Clustering::ClusterByPackage
  Root::Smalltalk::Clustering::ClusterByReference
  Root::Smalltalk::Clustering::ClusterByRefinedClusters
  Root::Smalltalk::Clustering::ClusterBySameExpression
  Root::Smalltalk::Clustering::ClusterByStronglyConnectedBase
  Root::Smalltalk::Clustering::ClusterByStronglyConnectedColla
  Root::Smalltalk::Clustering::ClusterRealOrStub
```

```
Reason Classes
Root::Smalltalk::Clustering::ClusterByClassHierarchy
Root::Smalltalk::Clustering::ClusterByDeltaIC
Root::Smalltalk::Clustering::ClusterByDominatingTrees
Root::Smalltalk::Clustering::ClusterByInterface
Root::Smalltalk::Clustering::ClusterByNamespace
Root::Smalltalk::Clustering::ClusterByPackage
Root::Smalltalk::Clustering::ClusterByReference
Root::Smalltalk::Clustering::ClusterByRefinedClusters
Root::Smalltalk::Clustering::ClusterBySameExpression
Root::Smalltalk::Clustering::ClusterByStronglyConnectedBaseComponents
Root::Smalltalk::Clustering::ClusterByStronglyConnectedCollapsedCompor
Root::Smalltalk::Clustering::ClusterRealOrStub
```

(a)



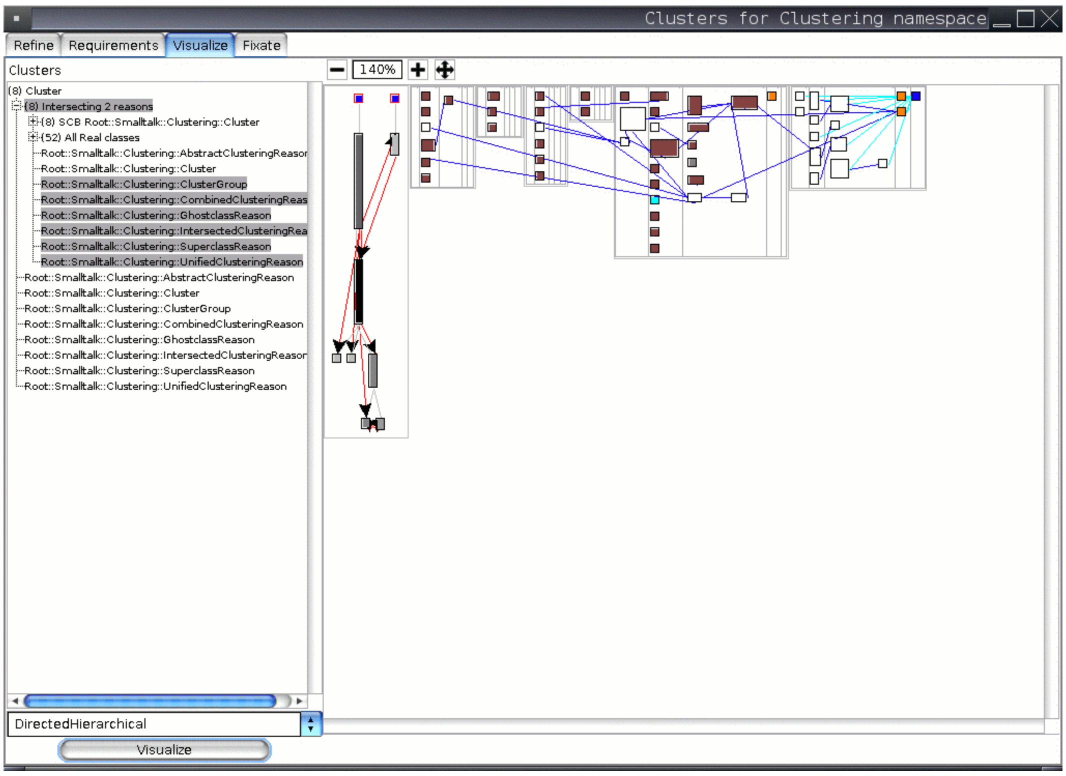
```
Clusters
(8) Cluster
(8) Intersecting 2 reasons
  (8) SCB Root::Smalltalk::Clustering::Cluster
  (52) All Real classes
   Root::Smalltalk::Clustering::AbstractClusteringReasor
   Root::Smalltalk::Clustering::Cluster
   Root::Smalltalk::Clustering::ClusterGroup
   Root::Smalltalk::Clustering::CombinedClusteringReas
   Root::Smalltalk::Clustering::GhostclassReason
   Root::Smalltalk::Clustering::IntersectedClusteringRea
   Root::Smalltalk::Clustering::SuperclassReason
   Root::Smalltalk::Clustering::UnifiedClusteringReason
 Root::Smalltalk::Clustering::AbstractClusteringReason
 Root::Smalltalk::Clustering::Cluster
 Root::Smalltalk::Clustering::ClusterGroup
 Root::Smalltalk::Clustering::CombinedClusteringReason
 Root::Smalltalk::Clustering::GhostclassReason
 Root::Smalltalk::Clustering::IntersectedClusteringReasor
 Root::Smalltalk::Clustering::SuperclassReason
 Root::Smalltalk::Clustering::UnifiedClusteringReason
```

```
Superclasses
Cluster->Set
ClusterGroup->Dictionary
```

```
Referenced Classes
Root::Smalltalk::Clustering::DirectedEdge
Root::Smalltalk::Clustering::ReasonClass
Root::Smalltalk::CodeFoo::Color
Root::Smalltalk::Core::Dictionary
Root::Smalltalk::Core::Exception
Root::Smalltalk::Core::MessageNotUnderstood
Root::Smalltalk::Core::OrderedCollection
Root::Smalltalk::Core::Set
Root::Smalltalk::Mondrian::LineShape
Root::Smalltalk::SCG::Moose::FAMIXClass
Root::Smalltalk::UI::TreeModel
```

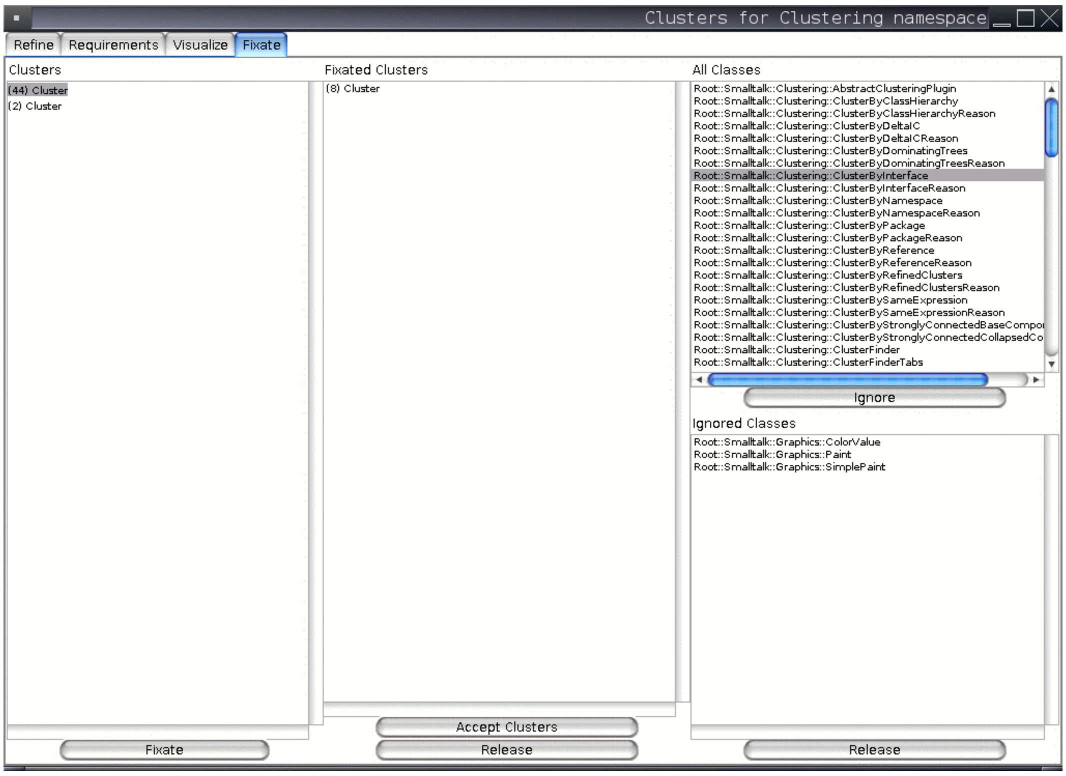
(b)

Figure A.1: *ClusterFinders* (a) *Main* and (b) *Requirements* interface



```
(8) Cluster
  (8) Intersecting 2 reasons
    (8) SCB Root::Smalltalk::Clustering::Cluster
    (52) All Real classes
    Root::Smalltalk::Clustering::AbstractClusteringReason
    Root::Smalltalk::Clustering::Cluster
    Root::Smalltalk::Clustering::ClusterGroup
    Root::Smalltalk::Clustering::CombinedClusteringReas
    Root::Smalltalk::Clustering::GhostclassReason
    Root::Smalltalk::Clustering::IntersectedClusteringRea
    Root::Smalltalk::Clustering::SuperclassReason
    Root::Smalltalk::Clustering::UnifiedClusteringReason
  Root::Smalltalk::Clustering::AbstractClusteringReason
  Root::Smalltalk::Clustering::Cluster
  Root::Smalltalk::Clustering::ClusterGroup
  Root::Smalltalk::Clustering::CombinedClusteringReason
  Root::Smalltalk::Clustering::GhostclassReason
  Root::Smalltalk::Clustering::IntersectedClusteringReason
  Root::Smalltalk::Clustering::SuperclassReason
  Root::Smalltalk::Clustering::UnifiedClusteringReason
```

(a)



Clusters:

```
(44) Cluster
(2) Cluster
```

Fixated Clusters:

```
(8) Cluster
```

All Classes:

```
Root::Smalltalk::Clustering::AbstractClusteringPlugin
Root::Smalltalk::Clustering::ClusterByClassHierarchy
Root::Smalltalk::Clustering::ClusterByClassHierarchyReason
Root::Smalltalk::Clustering::ClusterByDeltaIC
Root::Smalltalk::Clustering::ClusterByDeltaICReason
Root::Smalltalk::Clustering::ClusterByDominatingTrees
Root::Smalltalk::Clustering::ClusterByDominatingTreesReason
Root::Smalltalk::Clustering::ClusterByInterface
Root::Smalltalk::Clustering::ClusterByInterfaceReason
Root::Smalltalk::Clustering::ClusterByNamespace
Root::Smalltalk::Clustering::ClusterByNamespaceReason
Root::Smalltalk::Clustering::ClusterByPackage
Root::Smalltalk::Clustering::ClusterByPackageReason
Root::Smalltalk::Clustering::ClusterByReference
Root::Smalltalk::Clustering::ClusterByReferenceReason
Root::Smalltalk::Clustering::ClusterByRefinedClusters
Root::Smalltalk::Clustering::ClusterByRefinedClustersReason
Root::Smalltalk::Clustering::ClusterBySameExpression
Root::Smalltalk::Clustering::ClusterBySameExpressionReason
Root::Smalltalk::Clustering::ClusterByStronglyConnectedBaseCompo
Root::Smalltalk::Clustering::ClusterByStronglyConnectedCollapsedCo
Root::Smalltalk::Clustering::ClusterFinder
Root::Smalltalk::Clustering::ClusterFinderTabs
```

Ignored Classes:

```
Root::Smalltalk::Graphics::ColorValue
Root::Smalltalk::Graphics::Paint
Root::Smalltalk::Graphics::SimplePaint
```

(b)

Figure A.2: *ClusterFinders* (a) *Visualization* and (b) *Fixate* interface

Visualization Tab in figure A.2a is used to visualize candidate components in the *space of components*. Visualizations can be chosen by changing the value of the multiple choice field, labelled *DirectedHierarchical* in the figure. If clusters are selected when visualizing, they will be displayed using the selected visualization. If classes are selected, the *class blueprint* [Duc05] is automatically applied. When the selected visualization is a *component composition* visualization, it will be applied to all selected clusters, and the results will be shown next to each other. In case of an *interaction visualization*, the selected clusters will all be integrated in one visualization.

Fixate Tab allows users to add negative information. Classes from the globally available classes can be ignored manually. Classes in clusters which are considered complete can be ignored integrally by fixating their clusters. When a set of classes is ignored, the techniques are automatically recalculated. If the current *ClusterFinder* was opened using the *Find* button at the *Main Tab* in a parent *ClusterFinder*, fixated clusters can be accepted. This results in adding the fixated clusters to the *space of components* of the parent *ClusterFinder*.

Bibliography

[Are05] Gabriela B. Arevalo. *High Level Views in Object Oriented Systems using Formal Concept Analysis*. PhD thesis, Bern, Switserland, January 2005.

[BCL$^+$06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11–12):1257–1284, 2006.

[Ber83] Jacques Bertin. *Semiology of Graphics*. University of Wisconsin Press, 1983.

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[BT04] Markus Bauer and Mircea Trifu. Architecture-aware adaptive clustering of oo systems. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 3, Washington, DC, USA, 2004. IEEE Computer Society.

[CD00] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[CI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Softw.*, 7(1):13–17, 1990.

[cin] Cincom smalltalk^{TM}. On The Web. `http://smalltalk.cincom.com`.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 - the FAMOOS information exchange model. Technical report, 2001.

[Duc05] Stéphane Ducasse. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Softw. Eng.*, 31(1):75–90, 2005. Member-Michele Lanza.

[eAPS00] Fernando Brito e Abreu, Gonçalo Pereira, and Pedro Sousa. A coupling-guided cluster analysis approach to reengineer the modularity of object-oriented systems. In *CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering*, page 13, Washington, DC, USA, 2000. IEEE Computer Society.

[HC01] George T. Heineman and William T. Councill, editors. *Component-based software engineering: putting the pieces together.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling dynamic component interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *TOOLS 26, Technology of Object-Oriented Languages and Systems, Seite 58-70.* IEEE Computer Society, 1998.

[JCIR01] Hemant Jain, Naresh Chalimeda, Navin Ivaturi, and Balarama Reddy. Business component identification - a formal approach. In *EDOC '01: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[KC04] Soo Dong Kim and Soo Ho Chang. A systematic method to identify software components. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 538–545, Washington, DC, USA, 2004. IEEE Computer Society.

[Kos02] R. Koschke. Atomic architectural component recovery for program understanding and evolution. In *ICSM '02: Proceedings of the International Conference on Software Maintenance (ICSM'02)*, page 478, Washington, DC, USA, 2002. IEEE Computer Society.

[KP96] Christian Kramer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse*

Engineering (WCRE '96), page 208, Washington, DC, USA, 1996. IEEE Computer Society.

[LLSW03] Eunjoo Lee, Byungjeong Lee, Woochang Shin, and Chisu Wu. A reengineering process for migrating from an object-oriented legacy system to a component-based system. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 336, Washington, DC, USA, 2003. IEEE Computer Society.

[LSLW05] Eunjoo Lee, Woochang Shin, Byungjeong Lee, and Chisu Wu. Extracting components from object-oriented system: A transformational approach. *IEICE - Trans. Inf. Syst.*, E88-D(6):1178–1190, 2005.

[LYC$^+$99] Sang Duck Lee, Young Jong Yang, Eun Sook Cho, Soo Dong Kim, and Sung Yul Rhew. Como: A uml-based component development methodology. In *APSEC '99: Proceedings of the Sixth Asia Pacific Software Engineering Conference*, page 54, Washington, DC, USA, 1999. IEEE Computer Society.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: an agile information visualization framework. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[MS00] Ludger Martin and Elke Siemon. Component visualization based on programmer's conceptual models (poster session). In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 73–74, New York, NY, USA, 2000. ACM Press.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gĭrba. The story of moose: an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, 30(5):1–10, 2005.

[Sha53] Claude E. Shannon. Communication theory - exposition of fundamentals. *IEEE Transactions on Information Theory*, 1:44–47, 1953.

[STS99] Vijayan Sugumaran, Mohan Tanniru, and Veda C. Storey. Identifying software components from process requirements using domain model and object libraries. In *ICIS '99: Proceeding of the 20th international conference on Information Systems*, pages 65–81, Atlanta, GA, USA, 1999. Association for Information Systems.

[SW89] J. A. Simpson and E. S. C. Weiner. *The Oxford English Dictionary, Second Edition*. Oxford University Press, 1989.

[THCS01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter 21: Data structures for Disjoint Sets, pages 498–524. MIT Press and McGraw-Hill, 2001.

[TM01] A. Taivalsaari and I. Moore. *Prototype-Based Object-Oriented Programming: Concepts, Languages, and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2001.

[Tri01] A. Trifu. Using cluster analysis in the architecture recovery of object-oriented systems. Master's thesis, University of Timisoara, September 2001.

[Wei99] E. W. Weisstein. Venn diagrams. MathWorld–A Wolfram Web Resource, 1999. `http://mathworld.wolfram.com/VennDiagram.html`.

[WH92] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. *IEEE Trans. Softw. Eng.*, 18(12):1038–1044, 1992.

[WSY+06] Xinyu Wang, Jianling Sun, Xiaohu Yang, Chao Huang, Zhijun He, and Srinivasa R. Maddineni. Reengineering standalone c++ legacy systems into the j2ee partition distributed environment. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 525–533, New York, NY, USA, 2006. ACM Press.