Combining Clustering and Pattern Detection for the Reengineering of Component-based Software Systems

Markus von Detten
mvdetten@uni-paderborn.de

Steffen Becker
steffen.becker@uni-paderborn.de

Software Engineering Group & Heinz Nixdorf Institute
Department of Computer Science
University of Paderborn, Paderborn, Germany

ABSTRACT

During the software lifecycle, software systems have to be continuously maintained to counteract architectural deterioration and retain their software quality. In order to maintain a software it has to be understood first which can be supported by (semi-)automatic reverse engineering approaches. Reverse engineering is the analysis of software for the purpose of recovering its design documentation, e.g., in form of the conceptual architecture. Today, the most prevalent reverse engineering approaches are (1) the clustering-based approach which groups the elements of a given software system based on metric values in order to provide an overview of the system and (2) the pattern-based approach which tries to detect pre-defined patterns in the software which can give insight about the original developers' intentions. In this paper, we present an approach towards combining these techniques: we show how the detection and removal of certain bad smells in a software system can improve the results of a clustering-based analysis. We propose to integrate this combination of reverse engineering approaches into a reengineering process for component-based software systems.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.8 [**Software Engineering**]: Metrics—*product metrics*; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Documentation, Measurement

Keywords

Reengineering, Software Architecture, Clustering, Bad Smell Detection, Metrics

QoSA+ISARCS'11, June 20–24, 2011, Boulder, Colorado, USA.

1. INTRODUCTION

One of the most important and also most expensive phases in the life cycle of a software system is its maintenance. It is estimated that its cost can amount to as much as 80% of the total cost of the system [27]. Typical maintenance tasks such as the correction of bugs or the adaptation of functionality to new requirements slowly causes the architecture of the system to deteriorate. The software ages [23]. To counteract this aging effect, the software architecture has to be renovated and restructured constantly [16, 9]. In some cases, even the migration of a legacy system to a new paradigm like Service-Oriented Architecture is required (e.g., [32]).

In order to perform these maintenance tasks, a reengineer first has to understand the design of the software. Due to incomplete or outdated design documentation, this can be a tedious and very complicated task. Reverse engineering aims to analyze the source code of a software system automatically in order to recover its documentation [7]. In the past, two general types of approaches for the reverse engineering of software systems have been proposed: clustering-based reverse engineering and pattern-based reverse engineering [26].

Clustering-based reverse engineering approaches try to recover a system's architecture by grouping its basic elements (throughout this paper we will refer to these elements as classes) based on the values of certain code metrics. Examples of common metrics used for this purpose are coupling and stability [18]. The result of a clustering-based analysis is a set of components such as subsystems or modules and their connectors, i.e. the relationships between them [29]. These connected components represent a possible system architecture. Because the calculation of metric values is relatively easy for most metrics, a clustering-based analysis can be carried out in short time frames, even for large systems. A major drawback of this approach, however, is the high level of abstraction that comes with the use of metrics. Some complex architecture-relevant information cannot easily be captured by metric formulas. In addition, this approach can only recover the structure of the components but not their purpose. The reverse engineer has to manually inspect the elements in the detected components to get a feeling for what the component is supposed to do.

Pattern-based reverse engineering approaches aim at the detection of pre-defined structural or behavioral patterns in the software. These pre-defined patterns exist for different levels of abstraction, different domains and purposes but the most famous collection of patterns are the *Design Patterns* by Gamma et al. [13]. The rationale behind their detection is

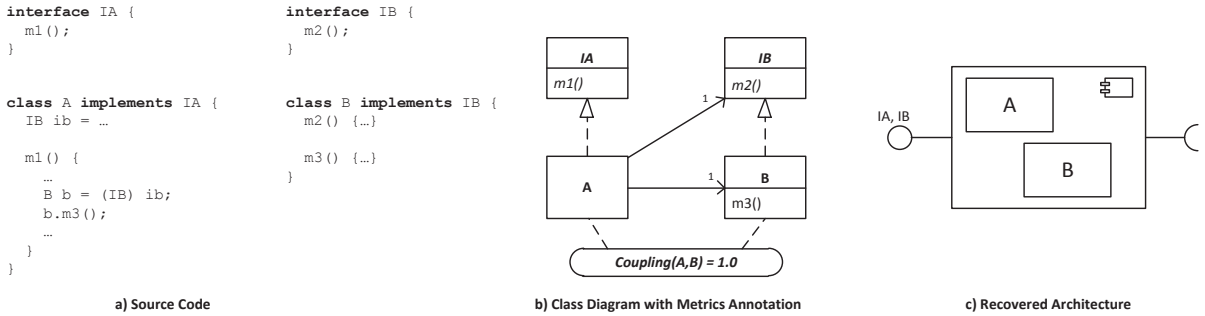

Figure 1: Example of the bad smell *Interface Violation* affecting the clustering

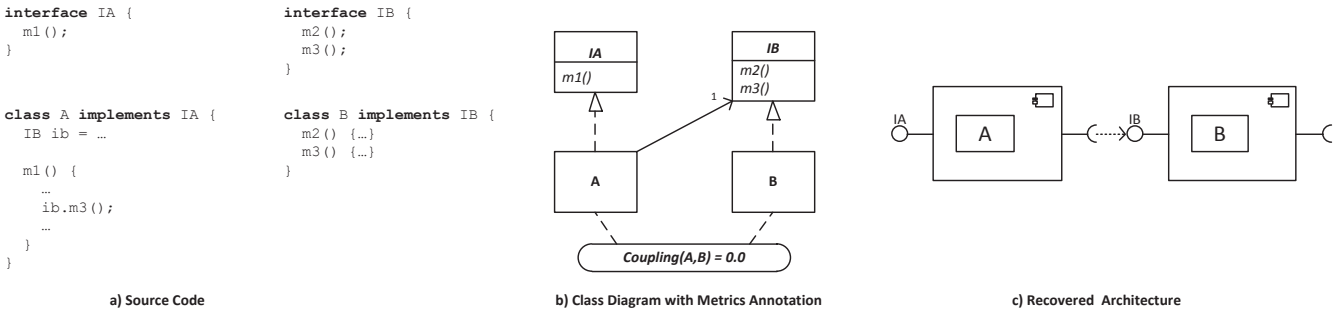

Figure 2: Refactored version of the example from Figure 1

that each design pattern has a specific intent that describes in which situation the pattern should be applied. Hence, the detection of a design pattern implementation in combination with the pattern's intent can reveal the developer's design rationale and thereby foster the understanding of the software. Another renowned category of patterns are design deficiencies like Anti Patterns [6] or Bad Smells [12] which describe frequently occurring bad solutions to common problems together with refactorings to correct them. Because pattern detection often takes much more detailed information like control and data flow into account than clustering-based approaches, it is generally much slower. Moreover, due to their usually low level of abstraction, an impractically large set of pattern implementations can be detected even for medium-scale systems.

The contribution of this paper is an approach for the combination of clustering-based and pattern-based reverse engineering techniques. It is an extension of the approach that was sketched in [30]. We show how the occurrence of bad smells can adulterate the architecture recovered by clustering-based reverse engineering approaches. We explain how these bad smells can be detected in the architecture and we sketch the integration of the reverse engineering step into an iterative reengineering process for component-based systems.

The paper is organized as follows. In Section 2, we illustrate the influence of bad smells on clustering-based reverse engineering approaches in general. Section 3 sketches the iterative reengineering process. It also explains the combination of clustering-based and pattern-based reverse engineering techniques to identify these bad smells. Examples of those bad smells are discussed in Section 4 while Section 5 presents early validation results for the bad smell detection. Section 6 gives an overview of related work and Section 7 concludes the paper.

2. PROBLEM

The reengineering process for software systems consists of reverse engineering and forward engineering. During the reverse engineering part, the software is analyzed, understood, and problems are identified. The forward engineering phase is concerned with changing the system, i.e., removing flaws, restructuring, or adding functionality [9].

At the beginning of the reengineering activity, in many cases, the reengineer will use clustering-based reverse engineering to get a first overview of the system. The metrics used by clustering-based reverse engineering approaches range from rather simple metrics like Coupling and Cohesion [18] to more complex, aggregated metrics such as Slice Layer Architecture Quality [8]. The measured metric values imply a certain clustering of the elements. For example, classes which are strongly coupled will probably be grouped together in the same component while uncoupled classes may be placed in different components.

The metric values used for the architecture recovery are based on the concrete architecture of the system, i.e., the architecture which is implemented in the existing source code. However, the concrete architecture is the result of a possibly long software lifecycle and may therefore have deteriorated from the originally intended, conceptional architecture. Maintenance and extensions may have introduced design flaws like bad smells into the code or may have led to other changes which are not in line with the conceptual architecture [23].

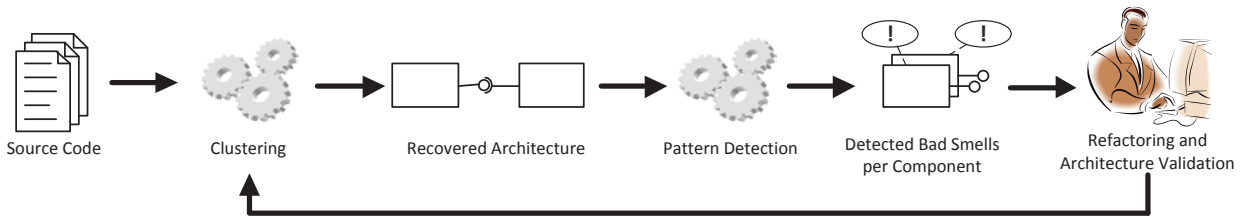

Figure 3: Sketch of the proposed reengineering process

Many bad smells lead to a high coupling of system elements. In this they contrast with design patterns, which for the most part strive to produce a modular, uncoupled architecture. Because bad smell occurrences in a system lead to strongly coupled classes they will cause them to be clustered together by an architecture recovery approach. Although they may not belong together conceptually, the classes are assigned to the same component due to the bad smell occurrence.

As an example, consider part a) of Figure 1: The classes A and B implement the interfaces IA and IB, respectively. Following the design principle "Program to an interface, not an implementation" [13], the classes are expected to communicate through their interfaces. However, A calls the method m3() from B. Because m3() is not provided by the interface IB, A downcasts the object ib to the concrete type B in order to access m3(). This intentional bypassing of an interface is called *Interface Violation*. It leads to a direct coupling of A and B as shown in the class diagram in part b). The coupling between A and B is calculated as the ratio between the number of accesses from A to B and the number of accesses from A to all classes in the system. In this simple example, the metric evaluates to $Coupling(A, B) = \frac{1}{1} = 1.0$ (Note that the Coupling metric does only take classes into account but not interfaces).

For a clustering-based analysis, the high coupling between A and B is a strong indicator to place the two classes together in one component as shown in part c) of Figure 1. The downcast of the object ib to the concrete type B is an example for the occurrence of a bad smell. It may have been introduced by an inexperienced developer who did not know that the classes were not supposed to communicate directly. It is also possible that the declaration of m3() was omitted from the interface IB by an oversight.

The bad smell in the example can easily be removed by adding the declaration of m3() to the interface IB. This leads to the situation depicted in Figure 2 a). Since m3() is now declared in IB, the downcast to B becomes unnecessary. There is no direct reference from A to B, anymore. Thus, the coupling metric for the two classes evaluates to 0 (Figure 2 b)) and the clustering algorithm would recover an architecture similar to the conceptual architecture in Figure 2 c).

This simple but illustrative example shows that the presence of bad smells in a system can adulterate the metric values used by the clustering and thereby can gradually obfuscate the original, conceptual architecture. Removing the occurrences from the code would improve the implemented, concrete architecture and lead to a better modularization of the system.

3. REENGINEERING PROCESS

In this section, we describe a process for the iterative reengineering of software architectures. It combines clustering-based and pattern-based reverse engineering techniques. In Section 3.1, we give an overview of the complete reengineering process. Section 3.2 deals with the analytic part of the process and explains our combination of reverse engineering techniques in detail.

3.1 An iterative reengineering process for component-based software architectures

In Section 2, we showed that bad smells can have a significant influence on the architecture recovered by clustering-based reverse engineering approaches. Thus, it stands to reason that bad smells should be detected and removed before a clustering is executed. This can be accomplished by a pattern detection which searches for a number of pre-defined bad smells and a subsequent manual removal of the detected bad smells.

However, because bad smells are usually defined on a rather low abstraction level, a pattern detection on a complete system can take a long time and yield a large number of results [31]. Therefore, we propose the approach which is sketched in Figure 3.

The starting point of the process is the source code of the software system. In a first step, the system is clustered to obtain an initial architecture. On the one hand, this can give the reengineer a first impression of the analyzed system. On the other hand, it allows the detection of bad smells for each of the initially inferred components. In contrast to a pattern detection in the complete system, the detection per component reduces the search space and thus the run time of the pattern detection.

In addition, the reengineer can focus the pattern detection on a subset of all detected components. If the initial clustering, for example, recovers an especially large component, this could be a promising starting point for a pattern detection because the removal of bad smells in this component might reveal a possibility to break up the component. Hence, the initial clustering enables the reengineer to manually steer the process in a direction which seems to promise the best results.

After the pattern detection step, the reengineer has to de-

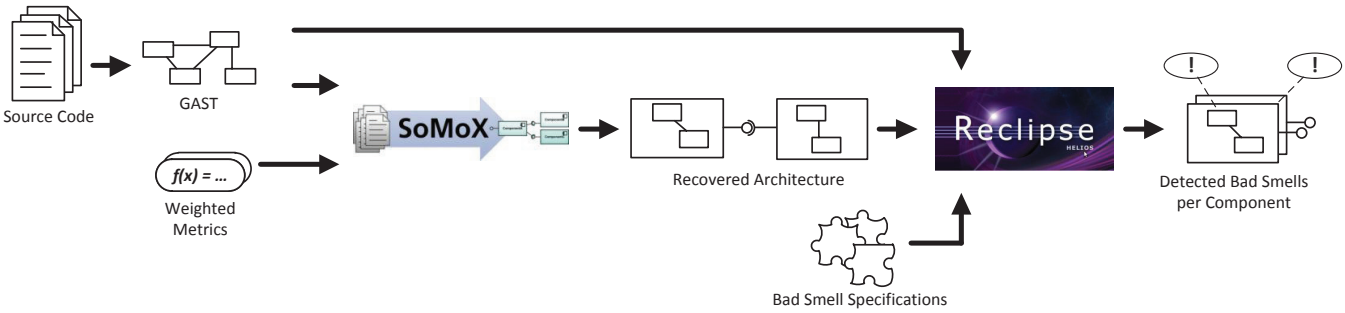

Figure 4: Combination of clustering and pattern detection as part of the reengineering process

cide which occurrences of bad smells have to be removed and how this can be accomplished. Some bad smell occurrences may be more critical than others which depends heavily on the context in which the bad smell occurs. For example, an Interface Violation (cf. Section 2) between two classes in the same component may be acceptable while it would be a major problem between two classes from completely different parts of the system (e.g., business logic and persistence layer). At this stage of our research, the reengineer has to inspect the bad smell occurrences manually and decide what to do with them on a case-by-case basis. An automated relevance analysis of the detected bad smell occurrences and support for their automated refactoring is subject to future work.

After the reengineer has refactored the critical bad smell occurrences, the system can be clustered again. Because it now contains fewer bad smells which can affect the clustering, the recovered architecture may contain different (usually smaller) components than the initial architecture. The reengineer may then focus his attention on another part of the system and try to detect and remove bad smells there.

Each iteration of the process improves both, the quality of the concrete system architecture (because bad smells are removed) and the reengineer's understanding of the system (because the clustering result reflects the improved architecture). Note that only the first clustering and pattern detection is real reverse engineering because the following iterations are already based on the refactored system. However, the subsequent iterations also improve the reengineer's understanding of the system because they reveal a clearer, albeit reengineered, architecture.

3.2 Combination of clustering and bad smell detection

This section describes in more detail how the combination of clustering and pattern detection is realized in our approach. Figure 4 shows an overview of this part of the process which was first sketched in [30].

For the clustering, we use the tool SoMoX (Software Model Extractor) which was developed at the FZI Karlsruhe [5]. The pattern detection is carried out by the tool Reclipse which was developed at the University of Paderborn [33, 34, 35].

Clustering with SoMoX.

SoMoX and Reclipse do not analyze the source code of the software directly but instead process a graph representation, the *generalized abstract syntax tree* (GAST). SoMoX calculates a number of metrics such as name resemblance, coupling, cohesion, and slice-layer architecture quality for the elements of the GAST. These metric values are weighted and aggregated to serve as input for the clustering algorithm. If the aggregated metric value for two elements exceeds a certain threshold, the two elements are clustered together in the same component. One important indicator for the clustering is the coupling of two elements.

The component definition that forms the basis for the clustering in SoMoX is in line with [28]. That means, e.g., that components are solely communicating via their interfaces, that interfaces define a number of services, and that services use data type arguments only.

The clustering algorithm starts by clustering the low-level system elements (e.g., classes) into basic components. It then proceeds to calculate new metric values for the detected basic components and continues to cluster them together into composite components. New metric values for the composite components are calculated, etc., until a stable architecture is achieved, i.e., until no combination of detected components has an aggregated metric value that exceeds the clustering threshold. The result of the clustering process is a mapping of the system elements to a number of reverse engineered components together with the connections between those components. SoMoX also allows to exclude certain classes from the clustering. This is useful when dedicated data classes like transfer objects or events are used in the system (see Section 4). For details on the metrics and the clustering refer to [8] and [17].

Pattern Detection with Reclipse.

The Reclipse Tool Suite [34] offers functionality for the specification and detection of structural and behavioral patterns. In this paper, we focus on the structural pattern specification and detection capabilities. Reclipse uses a graph matching approach for the detection of structural patterns [22]. The GAST of the software system is the host graph for this matching. The reengineer uses a graphical DSL to specify an arbitrary number of structural graph patterns which are to be detected by Reclipse.

Figure 5 shows a (slightly simplified) specification of the Interface Violation pattern introduced in Section 2. The specification shows the object structure that constitutes an occurrence of the pattern, i.e., a method call between two classes although the called class provides an interface. In the example, a Class c1 contains a Method m1 and a Class

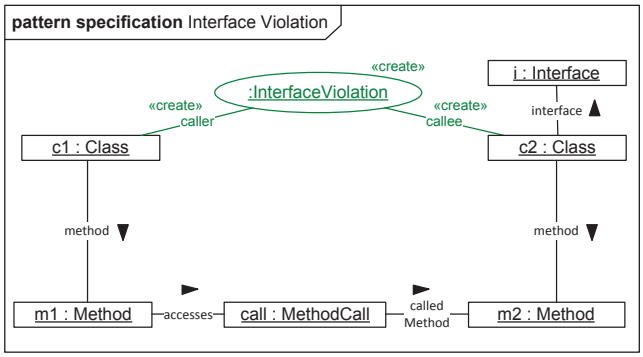

Figure 5: Structural specification of an Interface Violation pattern in Reclipse

c2 contains a Method m2. c2 also implements an interface i. The method m1 contains a method call. The called method is m2. This structural pattern can be enhanced further by constraints to ensure, e.g., that m2 is not an implementation of a method that was declared by the interface i. A more detailed explanation of the DSL and more example patterns can be found in [33] and [35].

When Reclipse detects a pattern occurrence in the host graph, i.e., the GAST, it creates an annotation (the green ellipse marked with ≪create≫ in Figure 5) which marks the pattern occurrence. The result of the pattern detection is a ranked list of bad smell occurrences. The ranking is computed automatically by relating the matched parts of the pattern to the total number of elements in the specification. For further details refer to [35].

Because we focus the pattern detection on the components recovered by SoMoX, the recovered architecture, i.e., the mapping of system elements to components, is also an input for the pattern detection (cf. Figure 4).

Note that a pattern detection which is carried out for each of the recovered components does not detect all patterns that a pattern detection run on the complete system would yield. Patterns whose elements are distributed over several components are missed by our combined detection approach. If, for some reason, the classes c1 and c2 from Figure 5 would be assigned to different components by the clustering, a component-by-component pattern detection would not detect the interface violation. However, the bad smells which we present in this paper increase the coupling of classes and therefore usually lead to classes being clustered together in large components. One primary goal of the reengineering approach is to break up these large components. Because the bad smells presented in this paper can successfully be detected within single components, we accept that patterns which are distributed across components are missed. In the future, we plan to extend the pattern detection from single components to pairs of components which are directly connected. This way, more patterns can be detected while unconnected components are not considered. Thus, the search space would still be smaller than for a pattern detection on the complete system.

4. EXAMPLE PATTERNS

In this section, we present several bad smells whose presence in a system increases the coupling between the involved classes and thereby can influence the clustering. We also point out possible refactorings to remove the bad smells.

4.1 Interface Violation

An example of the Interface Violation pattern was already shown in Section 2, so we only discuss the rationale behind the pattern here.

Components in a good component-oriented architecture are supposed to communicate exclusively via their interfaces [28]. Thus, if two classes are part of different components, they can only invoke operations of each other which are provided by their counterpart's interfaces. This leads to a good decoupling of the components. Only classes which reside in the same component can directly access each other's (public) operations and attributes which results in a high coupling. If classes from different components invoke operations which are not part of their corresponding interface, this is called an interface violation [8].

An existing interface violation has a direct impact on the clustering. Two classes which communicate with each other in a way that violates their interfaces are strongly coupled. Hence, they are probably assigned to the same component by a clustering-based reverse engineering approach.

A variant of this pattern is a direct access to a public attribute between two classes. This also bypasses the interface of the class that contains the attribute and creates a high coupling.

Refactoring.

There are different possibilities to remove an interface violation. The trivial solution would be to delete the violating method (or attribute) access. This would obviously reduce the coupling but also change the behavior of the system.

Another solution would be to extend the interface of the accessed class as shown in the example in Figure 2. This would decouple the two classes while preserving the behavior. On the other hand, all other classes which implement the same interface would have to be changed, too. In case of the interface violation variant by attribute access, one solution would be to provide an access method to the attribute and include that into the interface.

Third, a new interface that contains only the accessed method (m3() in Figure 1) could be introduced. This solution would avoid the deletion of behavior and the modification of existing interfaces.

4.2 Undercover Transfer Objects

Transfer objects are objects which serve as data containers for the communication between components. Transfer objects are "filled" with data by one component and then passed to another component which needs that data. Because they only serve as data containers, transfer objects are not really a part of the system architecture. Nevertheless, they are used by the sending and the receiving class and therefore may lead to a high coupling between them.

Due to their role as data containers which do not take part in the system behavior, transfer objects should be ignored by the clustering. Transfer objects are often specifically marked to be distinguishable from "normal" classes. The Common Component Modeling Example CoCoME [14] is an exemplary realization of a component-based trading system which we used for our validation (cf. Section 5). In

CoCoME, transfer objects are identified by the suffix TO and, thus, can be filtered out before the clustering. However, transfer objects which are not easily identifiable by a naming convention have to be detected and kept out of the clustering process.

The structure of transfer objects can be described by the bad smell Data Class [12]. Fowler describes data classes as classes which "have fields, getting and setting methods for the fields and nothing else". If this bad smell is discovered in a component, it may indicate that a transfer object was assigned to this component. This could mean that the classes which communicate by the this transfer object are also part of the component although they are not linked otherwise. Consequently, the transfer object should be excluded from the clustering in future iterations of the reengineering process.

4.3 Communication via Non-Transfer-Objects

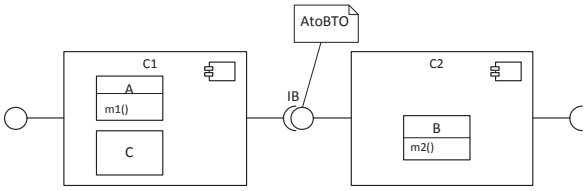

Figure 6: Conceptual architecture for communication via the transfer object AtoBTO

As defined by Szyperski [28] and pointed out in Section 4.2, two components should exchange data via transfer objects. Communication through objects which are not transfer objects should be avoided. For instance, instead of passing a certain object to another component to give the receiving component access to the object's data, a transfer object should be filled with the data.

Consider the conceptual architecture in Figure 6. The two components are connected via the interface IB and should exchange their data via the transfer object AtoBTO. In a flawed implementation, A could also pass a reference to C to B instead of populating a transfer object with data from C. This way, the coupling of A and B would be increased and B would have access to all the functionality of C although this is not intended by the conceptual architecture.

Refactoring.

To remove this bad smell, it is necessary to analyze the data that is really used by the called method. Afterwards, an appropriate transfer object can be prepared that is then used passed between the classes instead of the non-transfer-object. That way, the called method is prevented from accessing parts of the system that should be unknown to it.

4.4 Unauthorized Call

In contrast to an interface violation, an unauthorized call is an invocation of an operation which is provided by an interface but which is still prohibited because the calling component is not connected to the called component. In Figure 6, method m1() is allowed to call m2() provided m2() is part of interface IB and there are matching assembly and delegation connectors. On the other hand, m2() may not call m1() because component C2 does not require an interface from C1.

In programming languages which do not explicitly support the concept of components, e.g. plain Java, it is easy to introduce unauthorized calls accidentally because strict interface communication is not enforced by the language.

Refactoring.

Similar to the *Interface Violation* bad smell, the easiest solution is to delete the unauthorized call. However, in most cases the reengineer will have to analyze the reason for the unauthorized call. If, for example, a helper class is called, it may be a good idea to move that class to another component where it can be accessed by all classes that need it.If that is not possible, the needed functionality can either be sourced out into a separate component or it could be duplicated (although this obviously has other drawbacks).

Bad Smell	Detected occurrences	True positives
Interface Violation	11	11
Undercover Transfer Object	26	8
Non-TO Communication	32	7
Unauthorized call	3	3

Table 1: Validation results for CoCoME

5. VALIDATION

To validate our approach, we tried to detect the bad smells presented in Section 4 in the reference implementation[1] of the Common Component Modeling Example CoCoME [14]. CoCoME describes the architecture of a component-based trading system and is intended to illustrate good component-oriented design. The conceptual architecture is well-documented by use cases, component diagrams and sequence diagrams. The reference implementation was created manually by computer science students without component frameworks which makes it vulnerable to typical design flaws. The reference implementation consists of over 5000 lines of code in 127 classes.

5.1 Bad Smells in CoCoME

Table 1 shows the bad smell detection results for CoCoME. Column 1 shows the different bad smells introduced in Section 4. While the second column shows the number of occurrences that were reported by Reclipse, the third column shows the number of true positives among the detection results. The true positives were identified by manual inspection. We detected eleven *Interface Violations* and three *Unauthorized Calls* which were all true positives. In addition, eight *Undercover Transfer Objects* and seven *Non-TO Communications* were identified. The large number of false positives for the two patterns *Undercover Transfer Object* and *Non-Transfer Object Communication* can be explained by insufficient precision of our pattern specifications: Based on the detection results, we could improve the Reclipse spec-

[1]Available online at http://agrausch.informatik.uni-kl.de/CoCoME/downloads

ifications for these two patterns with further constraints in order to avoid most of the detected false positives.

Reclipse identified a number of classes as transfer objects which exhibited characteristic traits of transfer objects but were intended to model system events. Because they have a similar structure and purpose as the transfer objects, they should also be ignored by the clustering. Similar to transfer objects, the event classes can be filtered out by their suffix. So while they are false positives in the sense that they are no transfer objects, their detection is still valuable information for the reengineer. Similar to transfer objects, the system events can affect the clustering and this insight can be used to improve the preprocessing filters of SoMoX in subsequent iterations.

The Non-Transfer Object Communication pattern specifies a call with parameters which are not transfer objects. In our detection results, however, the pattern was also detected for parameter types from native Java libraries, e.g., java.util.Date. This could also be remedied by applying an appropriate filter.

In the following, we present details on the detected bad smell occurrences.

Figure 7 shows an excerpt from the conceptual architecture of CoCoME as documented in [14]. According to the documentation, the component TradingSystem::Inventory::Application contains the sub components Store and ProductDispatcher. Store can communicate with ProductDispatcher via the Interface ProductDispatcherIf. The component TradingSystem::Inventory::Data consists of the three sub components Enterprise, Persistence, and Store which are not supposed to communicate with each other. Store and ProductDispatcher from the Application component can access the Store sub component of the component Data via the interface StoreQueryIf. The sub components Enterprise and Persistence are used by other components which have been omitted in this example.

We detected 11 *Interface Violations* between the sub packages of the TradingSystem::Inventory::Data component: two between the subpackages Persistence and Enterprise and the other nine between Persistence and Store. This increases the coupling between the three sub components and prevents them from being distinguished clearly by the clustering.

We could identify eight *Undercover Transfer Objects*, i.e., data classes which only contain attributes and access methods. They were not marked with the name suffix "TO" although they are used as transfer objects. This is probably due to an oversight by the developers.

There are seven occurrences of the bad smell *Communications via Non-Transfer Objects* in the CoCoME reference implementation. Five of these communications take place between the components Tradingsystem::Inventory::Application::Store and Tradingsystem::Inventory::Data::Store. Two more occurrences of this bad smell were detected between the sub components ProductDispatcher and Store in the Application component.

We also detected three occurrences of the *Unauthorized call* bad smell. The subcomponent ProductDispatcher calls a method of the class FillTransferObjects in the sub component Store in two different places in the code. This is not allowed since Store requires the interface ProductDispatcherIf which is provided by ProductDispatcher but not the other way round. The same occurs between the sub components Tradingsystem::Inventory::Data::Store and Tradingsystem::Inventory::Application::Store. An inspection of the corresponding classes showed that the class FillTransferObjects from the sub component Store is a helper class that is also accessed from the sub component ProductDispatcher. This access was probably introduced when the developers realized that they needed the functionality of FillTransferObjects in the other component as well. Because Java does not enforce interface communication between components, the unauthorized call could be added unnoticed.

5.2 Discussion

Although our validation only is a first experiment on a comparably small system, it shows that the bad smells presented in this paper do occur in practice. The reference implementation of CoCoME was created manually and although the design documentation is very detailed and clear, numerous bad smells exist in the implementation. It stands to reason that this problem will be even more significant in larger, more complex systems.

A clustering-based analysis of CoCoME with SoMoX was already performed [8, 17]. It showed that 9 of the 14 documented components could be recovered. However, the subcomponent Tradingsystem::Inventory::Application::Store which is a part of several bad smell occurrences could not be recovered by SoMoX. This leads us to believe that the bad smells are actually responsible for SoMoX missing the sub component.

In addition to the reference implementation, there are a number of other CoCoME implementations which make use of different component frameworks like SOFA or FRACTAL [24]. These frameworks are intended to support the creation of a good component-oriented design, e.g., by prohibiting the external access to methods which are not part of a component's interface. In this context, it would be interesting to analyze these implementations and compare if they contain significantly fewer bad smells than the reference implementation and how this affects the clustering.

6. RELATED WORK

In this section, we present related work from different areas of research. First, we give a short overview of clustering-based and pattern-based reverse engineering techniques. Afterwards, we discuss approaches which combine different reverse engineering techniques to improve the detection results.

Clustering-based Reverse Engineering.

Architecture recovery is a wide field of research for which Ducasse and Pollet [11] attempt to present a taxonomy and give an overview of different approaches.

Müller et al. [21] were among the first to present an automated architecture recovery approach which groups basic system building blocks based on metrics like coupling and cohesion. Their approach is implemented in the Rigi tool.

Koschke [16] analyzes and compares different architecture recovery techniques and proposes a framework to combine the different approaches in order to achieve a better recovery result. The approaches used in the combination all are clustering-based. The way in which the different recovery approaches are combined is comparable to the combination of metrics used by SoMoX. Koschke suggests in his paper that more detailed and precise information than the one used by the clustering-based approaches should be taken into account in order to achieve even better recovery results.

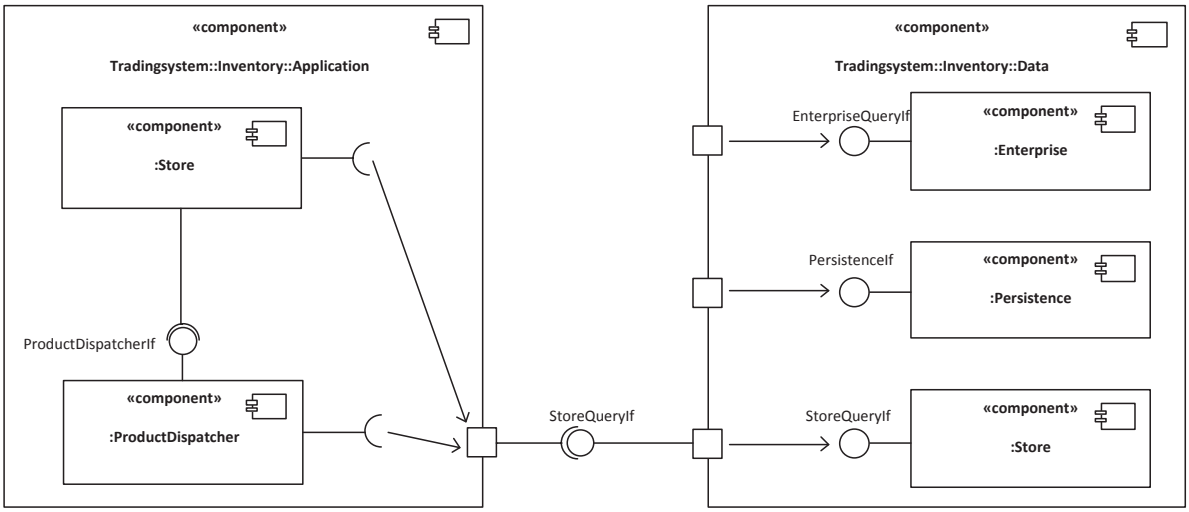

Figure 7: An excerpt from the conceptual architecture of CoCoME [14]

Pattern-based Reverse Engineering.

Similar to clustering-based reverse engineering, many approaches for detection of software patterns exist. Here we only discuss a few select publications which are close to our approach because they are concerned with components or the detection of design flaws. An exhaustive overview of different approaches in pattern-based reverse engineering is given by Dong et al. [10].

Keller et al. [15] describe an approach to detect "design components" in source code. However, they refer to a design component as "a package of structural model descriptions together with informal documentation, such as intent, applicability, or known-uses." Hence, they detect design *patterns* rather than components. In contrast, the components recovered by SoMoX [5] in our approach are in line with the component definition by Szyperski [28].

Sartipi [26] uses data mining techniques to structure a graph representation of a program and then defines architectural patterns (or, as he calls them, queries) on the resulting graph which are evaluated by graph matching. The queries are focused on simple architectural properties like the number of relations to a certain component and are not as expressive as the structural patterns used in our approach.

In [31], Trifu et al. detect and remove design flaws with respect to a user-selected quality goal, e.g., performance. They detect those flaws by using graph matching in combination with basic metrics. However, the authors point out that this leads to a large number of detection results which have to be manually validated by the user. In our approach, we propose to cluster the system before anti patterns are detected to speed up the detection and also reduce the number of results.

Munro [20] as well as Salehie et al. [25] use a combination of metrics to detect occurrences of different design flaws in a system. In contrast to our approach they do not use metrics to recover an architecture but employ the metric values as indicators for the existence of anti patterns.

Moha et al. [19] present an approach for the specification and detection of bad smells in code. They provide a method to derive consistent specifications from textual descriptions and generate detection algorithms from these specifications which take metrics values as well as structural aspects into account. However, they do not employ clustering-based techniques in their approach. They also do not consider the removal of bad smells but focus on their detection.

Arendt et al.[2] present a quality assurance process which uses a combination of metrics and structural patterns to identify model smells. They combine the bad smells with pre-defined graph transformations to provide an automated refactoring for identified bad smells. They use their approach for the quality assurance in an industrial context, so they assume that the analyzed models already exist and do not have to be recovered.

Combination of reverse engineering approaches.

Bauer and Trifu [4] use a combination of pattern detection and clustering to recover the architecture of a system. They detect so-called architectural clues with a Prolog-based pattern matching approach and use these clues to compute a multi-graph representation of the system. The weighted edges in this representation indicate the coupling of the system elements and are used by a clustering algorithm to obtain an architecture of the system. In contrast to our approach, the clustering is completely based on the information gathered by the pattern detection. Thus, the pattern detection has to be carried out first which can take very long for large systems. Our approach applies the clustering first to reduce the search space for the pattern detection. In addition, Bauer and Trifu focus on the detection of design patterns and do not consider the impact of bad smells on the clustering.

Basit and Jarzabek [3] identify clone patterns in programs and then apply a data mining approach to cluster clones which occur together frequently. However, they apply the clone detection and the clustering consecutively and do not consider relation between the two parts nor do they suggest

multiple iterations of their approach. The detection of predefined patterns is not addressed.

Similar to our approach, Arcelli Fontana and Zanoni [1] use an AST representation of a system as a common basis for pattern detection and architecture recovery. However, they use the two techniques in parallel but do not combine them.

7. CONCLUSION AND FUTURE WORK

In this paper, we presented an approach for the combination of clustering-based and pattern-based reverse engineering approaches. We showed that the presence of bad smells in the code of a software system can adulterate the results of a metric-reliant clustering and that it is therefore sensible to detect and remove the bad smell occurrences before the system is clustered. To mitigate the usually high run-time of the pattern detection and allow a more focused detection of bad smells, we also proposed to start with the clustering and thus obtain an initial decomposition of the system. Afterwards, the pattern detection is applied to each detected component to detect bad smells which prevent a further decomposition by the clustering algorithm.

As a starting point, we identified several bad smells that have an impact on the coupling metric which is a key indicator for the clustering. We also presented first experimental validation results for different component-based software systems which showed that the identified bad smells really occur in these systems.

In the future, we plan to extend our approach by a relevance analysis for bad smell occurrences which takes additional metric values into account. This will help to distinguish bad smells which should be removed from those which do not have a negative influence on the architecture. We also want to analyze, if and how the detection design patterns can be incorporated into our approach. It would also be interesting to examine which other metrics besides coupling are influenced by patterns. Finally, we plan to add tool support for the removal of bad smells which is done manually at the moment. Presenting the reengineer with sensible refactoring options can prevent the accidental introduction of new bad smells during the removal of existing ones.

8. ACKNOWLEDGMENTS

We would like to thank Oleg Travkin for his ideas, his conceptual work and his work on the prototype implementation.

9. REFERENCES

[1] F. Arcelli Fontana and M. Zanoni. A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324, 2011.

[2] T. Arendt, S. Kranz, F. Mantz, N. Regnat, and G. Taentzer. Towards Syntactical Model Quality Assurance in Industrial Software Development: Process Definition and Tool Support. In *Proceedings of the Software Engineering 2011*. Springer, 2011. to appear.

[3] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. *SIGSOFT Software Engineering Notes*, 30(5):156–165, September 2005.

[4] M. Bauer and M. Trifu. Architecture-Aware Adaptive Clustering of OO Systems. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pages 3 – 14, March 2004.

[5] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron. Reverse Engineering Component Models for Quality Predictions. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*, pages 199–202. IEEE Computer Society, 2010.

[6] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mombray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, Inc., 1998.

[7] E. J. Chikofsky and J. H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[8] L. Chouambe, B. Klatt, and K. Krogmann. Reverse Engineering Software-Models of Component-Based Systems. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pages 93–102, Athens, Greece, 2008. IEEE Computer Society.

[9] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, Switzerland, June 2008.

[10] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, 2009.

[11] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME - The Common Component Modeling Example. In *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer Berlin / Heidelberg, 2008.

[15] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21st International Conference on Software Engineering*, pages 226–235. IEEE Computer Society Press, May 1999.

[16] R. Koschke. Atomic Architectural Component Recovery for Program Understanding and Evolution. In *Proceedings of the International Conference on Software Maintenance*, pages 478 – 481. IEEE, 2002.

[17] K. Krogmann. *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2010. to appear.

[18] R. Martin. OO Design Quality Metrics - An Analysis of Dependencies. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 1994.

[19] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36:20–36, 2010.

[20] M. J. Munro. Product metrics for automatic identification of "bad smell" design problems in Java source-code. In *11th IEEE International Symposium on Software Metrics*, pages 9 pp. –9, Sept. 2005.

[21] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, 1993.

[22] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering*, pages 338–348. ACM Press, 2002.

[23] D. L. Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[24] A. Rausch, R. Reussner, R. Mirandola, and F. Plašil, editors. *The Common Component Modeling Example - Comparing Software Component Models*. Number 5153 in Lecture Notes in Computer Science. Springer, 2008.

[25] M. Salehie, S. Li, and L. Tahvildari. A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006)*, pages 159 –168. IEEE Computer Society, 2006.

[26] K. Sartipi. Software Architecture Recovery based on Pattern Matching. In *Proceedings of the International Conference on Software Maintenance*, pages 293–296. IEEE Computer Society, 2003.

[27] I. Sommerville. *Software Engineering*. Addison Wesley, 4th edition, May 1992.

[28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[29] R. N. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, Feb. 2009.

[30] O. Travkin, M. von Detten, and S. Becker. Towards the Combination of Clustering-based and Pattern-based Reverse Engineering Approaches. In *Proceedings of the 3rd Workshop of the GI Working Group L2S2 - Design for Future 2011*, Feb. 2011. to appear.

[31] A. Trifu, O. Seng, and T. Genssler. Automated Design Flaw Correction in Object-Oriented Systems. In *Proceedings of the 8th Euromicro Working Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 174–183. IEEE Computer Society, 2004.

[32] A. Umar and A. Zordan. Reengineering for service oriented architectures: A strategic decision model for integration versus migration. *Journal of Systems and Software*, 82:448–462, March 2009.

[33] M. von Detten, M. Meyer, and D. Travkin. Reclipse - A Reverse Engineering Tool Suite. Technical Report tr-ri-10-312, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Mar. 2010.

[34] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*, pages 299–300. ACM Press, 2010.

[35] M. von Detten and D. Travkin. An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw. Technical Report tr-ri-10-322, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Oct. 2010.