

Reengineering component-based software systems with Archimetric

Markus von Detten · Marie Christin Platenius ·
Steffen Becker

Received: 31 January 2012 / Revised: 28 March 2013 / Accepted: 2 April 2013 / Published online: 14 April 2013
© Springer-Verlag Berlin Heidelberg 2013

Abstract Many software development, planning, or analysis tasks require an up-to-date software architecture documentation. However, this documentation is often outdated, unavailable, or at least not available as a formal model which analysis tools could use. Reverse engineering methods try to fill this gap. However, as they process the system's source code, they are easily misled by design deficiencies (e.g., violations of component encapsulation) which leaked into the code during the system's evolution. Despite the high impact of design deficiencies on the quality of the resulting software architecture models, none of the surveyed related works is able to cope with them during the reverse engineering process. Therefore, we have developed the Archimetric approach which semiautomatically recovers the system's concrete architecture in a formal model while simultaneously detecting and removing design deficiencies. We have validated Archimetric on a case study system and two implementation variants of the CoCoME benchmark system. Results show that the removal of relevant design deficiencies leads to an architecture model which more closely matches the system's conceptual architecture.

Keywords Reengineering · Reverse engineering · Software architecture · Component-based software systems ·

Architecture reconstruction · Design deficiencies · Deficiency detection · Code metrics · CoCoME

1 Introduction

Long-living software systems (especially business information systems) tend to have large code bases with millions of lines of code. These code bases grew over decades often without taking special formalized models of the system's architecture into account. However, such models are necessary for a large diversity of activities which need software architecture models as input. These include manual activities like project management or planning the system's evolution as well as automated analyses of the software architecture like model-driven quality analyses (performance, scalability, reliability), and requirements tracing. To cope with the problem of missing architectural models, architecture reconstruction methods have been proposed in the past. They analyze the source code of the system and try to extract an architecture model of the current implementation. Most of these methods rely on the clustering of source code elements like classes and interfaces.

However, these architecture reconstruction methods are severely impacted by *design deficiencies* (also called design defects sometimes) in the system's code base. For example, if a class has an association with another class in the code base, which it should not have according to the conceptual architecture (i.e., the architecture of the system as intended by the software architect), then clustering methods may group those two classes erroneously. This implies that as long as design deficiencies are not accounted for explicitly in the architecture reconstruction method, software architects will experience a reduced quality of the reconstructed architecture, i.e., subsystems, component structures, and connectors

Communicated by Prof. Dr. Dorina Petriu and Dr. Jens Happe.

M. von Detten (✉) · M. C. Platenius · S. Becker
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Paderborn, Germany
e-mail: Markus.von.Detten@upb.de

M. C. Platenius
e-mail: Marie.Christin.Platenius@upb.de

S. Becker
e-mail: Steffen.Becker@upb.de

will be inaccurate. In order to improve the reconstruction of architectures, a method that takes design deficiencies into account is necessary.

Despite the outlined impact of design deficiencies on the quality of the reconstructed architecture, no existing architecture reconstruction approach explicitly integrates a systematic deficiency removal into its recovery process. There are, however, approaches which implement different combinations of architecture reconstruction, detection of bad code patterns, or refactoring and reengineering. Nevertheless, no approach combines all three areas yet.

To solve the outlined problems, we present our Archimatrix approach. Archimatrix is a tool-supported architecture reconstruction process. It enhances a clustering-based architecture reconstruction approach called SoMoX [16,33] with extensible design deficiency detection steps relying on (source code) pattern matching implemented in the Reclipse tool [61–63]. In this paper, we present a detailed explanation of both, the process to execute Archimatrix, as well as the process to extend Archimatrix with new design deficiency formalizations. For each step in Archimatrix, we provide a detailed conceptual and technical description. This work consolidates our Archimatrix process on the basis of a new running example and extends our previous works [42,55,59]. In contrast to them, this paper describes additional relevant process steps (e.g., for extending Archimatrix with new deficiency formalizations), an extended evaluation taking additional systems and implementations into account, and a deep and systematic-related work survey. At the same time, the paper summarizes a PhD thesis on this topic [58].

We validate our approach on two case studies. On the one hand, we applied our approach to the Palladio Fileshare application, a small case study application suited to illustrate our paper's running example. On the other hand, we analyzed and compared two variants of the Common Component Modeling Example (CoCoME) [27]. CoCoME has been designed by several research groups from the area of component-based software development as a benchmark system for the evaluation and comparison of different component-based analysis methods. It consists of a well-documented conceptual architecture (cf. Chapter 1 in [27]) and a reference implementation which is supposed to follow that conceptual architecture. Our results provide strong evidence that we are able to identify relevant deficiencies in both systems and that their removal leads to an increased quality of the reconstructed architectures. We consider an architecture model to be of a better quality than another if the reconstructed architecture is closer to the conceptual architecture of our case study systems.

The contribution of this paper is a consolidated and extended software architecture reconstruction process called Archimatrix. In this paper, we outline the process to use Archimatrix on a novel running example covering all aspects of Archimatrix and add a process for extending Archimatrix.

We extended previous validations with new systems and new implementation variants so that we can present results based on three different code bases of two different systems. The results show that we are able to detect and remove relevant design deficiencies in practice. Finally, we have extended our related work survey to highlight the novel ideas of Archimatrix.

This paper is structured as follows. The next section introduces our foundations and key assumptions. Section 3 then introduces the design deficiency “Transfer Object Ignorance” which we use as a running example throughout the paper. We outline the Archimatrix process in Sect. 4. We detail on individual steps of Archimatrix in Sects. 5–9. After presenting our evaluation results in Sect. 11, we discuss related work in Sect. 12 and conclude our paper by giving an outlook on future work.

2 Foundations and assumptions

In this section, we describe the foundations of this paper. First, we clarify our notion of software architecture and software components. Then, we describe the development paradigm that we assume in this paper and state our assumptions regarding the availability of architectural documentation. Finally, we classify our approach in terms of the architecture reconstruction taxonomy introduced in [19].

2.1 Software architecture and components

Our view of a software architecture in the context of this paper is centered on the static structure of a software system. A software architecture consists of a number of interconnected software components.

The term *software component* has been used in many approaches, projects, and standards; yet, there is no universally accepted definition of what a software component is.

The definition of a software component used in this paper is in line with the definition by Szyperski [51]: “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.”

The components Szyperski describes are typically used in business information systems. They have clearly defined interfaces and are solely communicating via those interfaces. Interfaces define a number of operations and operations exchange data only via so-called data transfer objects [3,24], i.e., objects whose classes are simple data records. These classes only have fields of primitive types like integer plus corresponding access methods. This leads to a clear decoupling of components and promotes component encapsulation.

sulation, interchangeability, and reuse. It also increases the maintainability of the architecture.

A component is an entity of the architecture level and therefore more abstract than entities on the implementation level, like classes. A component can comprise a number of implementation level entities. In that case, it is called *primitive component*. A component can also be recursively composed of other components in which case it is referred to as *composite component*.

Note that this notion of a component is different from the definition of components in other approaches like the one by Keller et al. [28] where each class is viewed as a component of the software. It also differs from the notion used in embedded systems where components are often active and communicate via asynchronous message passing.

2.2 Underlying development paradigm

We assume a scenario in which an object-oriented programming language is used to develop component-based systems. For the purpose of the examples in this paper and for the validation of the approach, we assume a system that was implemented in Java. However, the Archimatrix process in general is not dependent on the use of a concrete programming language as it operates at the level of a program's abstract syntax tree. Therefore, our approach can be applied to any language for which the software architect is able to extract the abstract syntax tree. Currently, the Archimatrix tool is able to create AST representations of Java, C++, and Delphi programs (see Sect. 10).

2.3 Availability of architectural documentation

For the reverse engineering scenario presented in this paper, we assume that no documentation of the software system under analysis is available. This is either the case when the documentation is so outdated that it has become useless or when it has not been created in the first place. This also implies that no formal models of the system exist. Therefore, the only reliable source of information about the system is the source code itself. However, for the validation of our approach, an architectural documentation is necessary to judge whether Archimatrix has a meaningful effect on the system's architecture.

2.4 Classification of the Archimatrix approach

Following the taxonomy by Chikofsky and Cross, Archimatrix is essentially a reverse engineering approach as its main goal is "analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction" [14]. However, Chikofsky and Cross

define that reverse engineering approaches are strictly non-invasive, i.e., the system under analysis may not be modified. For them, modifications are forward engineering activities. Since Archimatrix comprises both, reverse engineering in the form of architecture reconstruction and forward engineering in the form of deficiency removal, we designate it as a reengineering approach.

3 Running example

In this section, we introduce an example that is used to illustrate our approach for the remainder of the article. First, we show a simple component-based system. Then, we introduce one of the design deficiencies which might occur in such a system.

3.1 Example system

Figure 1 shows the conceptual software architecture of an example system. The architecture is intentionally simplified as the system is only used for illustrative purposes. It represents a store management system similar to the CoCoME system which was used in the validation. The system consists of eight components which are named according to their purpose: logistics, payment, accounting, controlling, store, database, UI, and network. The components are connected via a number of interfaces.

This conceptual architecture could serve as a basis for the implementation of such a store system. If the code was then analyzed by an architecture reconstruction approach, this architecture would be the expected result. However, if this is actually the case depends on the implementation. For example, the presence of design deficiencies in the implementation could lead the architecture reconstruction to a different result. One such design deficiency is explained in the following section.

3.2 Design deficiency: Transfer Object Ignorance

In this section, the Transfer Object Ignorance deficiency is presented. It is an example of an architecturally relevant deficiency whose presence can impact the architecture reconstruction. Besides the Transfer Object Ignorance, we formalized three other deficiencies: Interface Violation, Unauthorized Call, and Inheritance Between Components. These deficiencies were also used in the validation of our approach and are explained on the Archimatrix Web site [1]. New deficiencies which also represent a neglect of component-oriented design principles can be added in Archimatrix. We describe a process for this in Sect. 4.2.

To describe the Transfer Object Ignorance deficiency, we use a template that is similar to the Mini-AntiPattern template

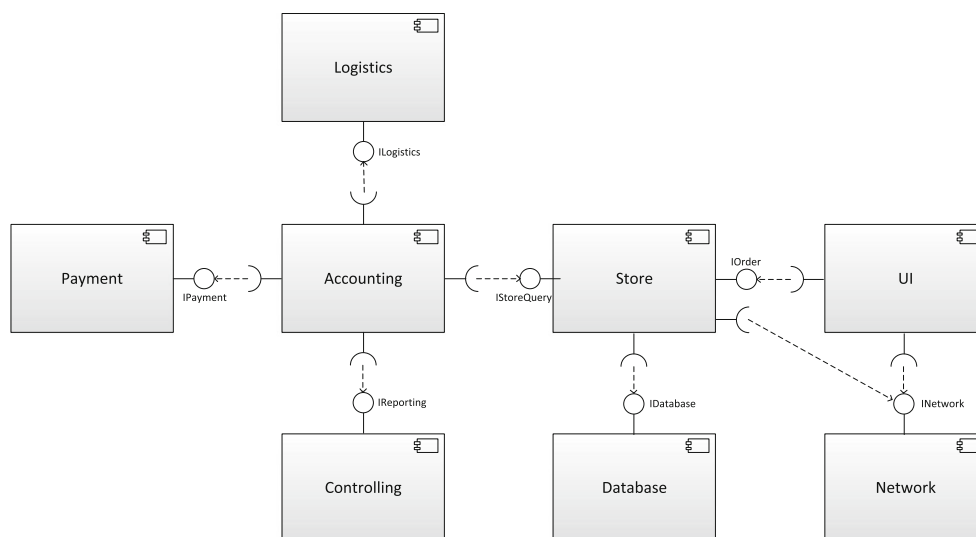


Fig. 1 Example of a simple trading system

by Brown et al. [13]. The template states the name of the deficiency, the problem that is created by it, gives an example, and presents a number of strategies to remove the deficiency.

Name Transfer Object Ignorance

Design Deficiency Problem In component-based or service-oriented architectures, two components or services should exchange data only via data transfer objects [3,24]. A transfer object is a data record that contains only the data that are needed for a specific task and no additional behavior. The only methods of a transfer object are access methods for the contained data. In contrast to common practice in object-oriented programming, communication by exchanging references to objects which are not transfer objects should be avoided. This has two main reasons: First, passing an object reference to another component in order to allow access to that object's data is a security risk and breaks the sending component's encapsulation. It inadvertently offers the receiving component the opportunity to invoke arbitrary methods of the exposed object. Second, as each component can be deployed independently [51], each method call may possibly be a remote call that incurs a significant communication overhead. Polling data by invoking a number of getters on a remote component can therefore be inefficient [3,24]. Thus, transfer objects should be used that can be filled with the relevant data and then be sent to the receiving component as a whole.

Example Figure 2 shows the three components Accounting, Controlling, and Store from the example system. Inside the components, examples of classes that were assigned to them are shown. Interfaces to other components are omitted in this

figure. Let us assume the convention that in this system, all transfer objects should be marked by appending the suffix "TO" to their names. This allows SoMoX to recognize them as transfer object and ignore their coupling in the architecture reconstruction. The system contains two Transfer Object Ignorance occurrences which are marked by labeled ellipses. Occurrence no. 1 is between the components Accounting and Store while occurrence no. 2 is between Accounting and Controlling.

The Accounting component contains a class Assets which is able to calculate the total value of the items that are currently in store. To do this, the method calculateValue accesses the method getInventory of the IStoreQuery interface. In the example, this method returns an instance of the class Inventory which is then used to calculate the value of the stored items. This behavior of the example system constitutes an occurrence of the Transfer Object Ignorance design deficiency. By receiving an instance of Inventory, the Accounting component cannot only access the data contained in the inventory object but it could also add and remove items from the inventory and call non-accessor methods like checkStock. Instead, the class StoreQuery should create a transfer object, populate it with data from the Inventory, and pass it to the Accounting component.

Occurrence no. 2 is another variant of the Transfer Object Ignorance deficiency. Instead of exposing an object by returning it in response to a call, a non-transfer object is passed as a parameter. The Accounting component can send a Report to the Controlling component by calling the method sendReport of the class Reporting. Here, the Report object is passed to another component. In this simple example, it only contains one field of type string which stores the report's contents. Although Report looks very much like a transfer object, it lacks the appropriate suffix "TO."

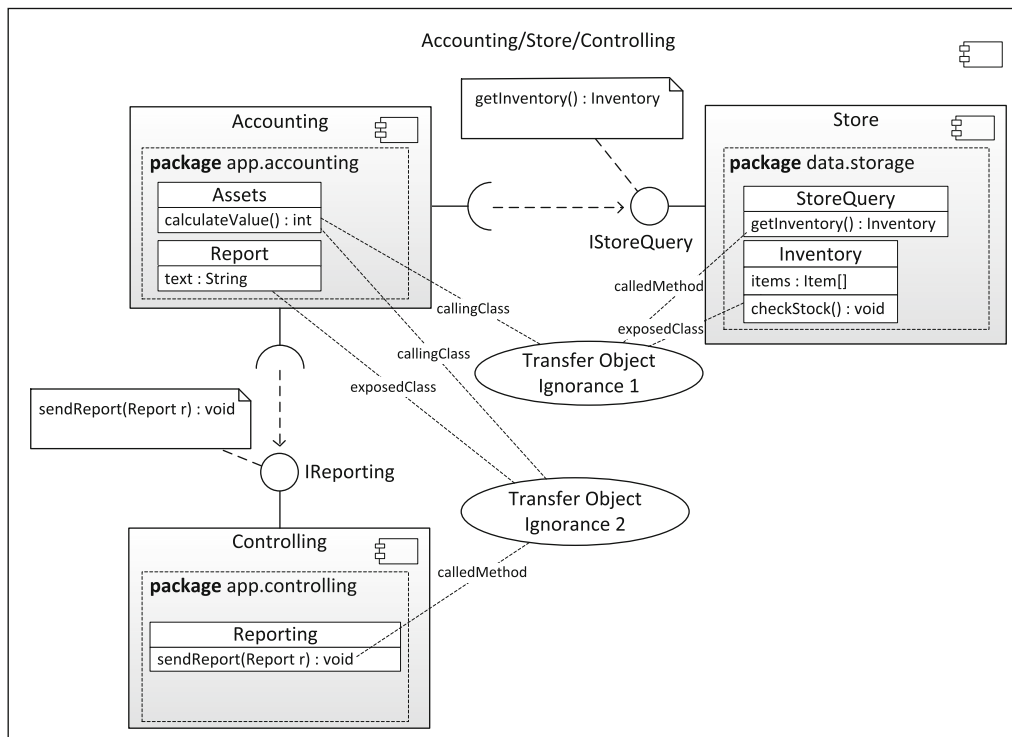


Fig. 2 Transfer Object Ignorance occurrences in the example system

Removal strategy In order to remove the first Transfer Object Ignorance deficiency, the functionality could be moved somewhere else. Calculating the total value of all the stored items should be the responsibility of the Store component. So the `calculateValue` method could be moved to the class `StoreQuery` in the Store component. Afterward, it could be added to the interface `IStoreQuery` and all calls in the Accounting component could be redirected to this new interface method. This way, only the calculated total value would have to be sent to the Accounting component.

The other deficiency could simply be removed by adding the suffix “TO” to the `Report` class and thereby marking it explicitly as a transfer object.

Note that every removal strategy has its pros and cons. Moving the behavior and extending the interface as described above removes the occurrence of Transfer Object Ignorance deficiency no. 1. But extending the interface of course has an influence on every class which implements that interface. The software architect has to keep that in mind when applying this strategy.

Another possibility to remove this deficiency is to create a new data class for the transfer object, e.g., `InventoryTO`. This would need attributes for all relevant data that are required by the `Assets` class. To find out which data are required, the software architect would have to analyze the `calculateValue` method. Afterward, the method `getInventory` would have to be adapted, to create an instance of `InventoryTO`, populate it with the necessary data, and return it instead of the `Inventory`

instance itself. Because analyzing which data are used by the Accounting component is very complex, it is hard to create an automatic transformation that performs this reengineering.

3.3 Consequences of the design deficiency

If the example system was subject to a clustering-based architecture reconstruction algorithm, the Transfer Object Ignorance deficiencies could impact the reconstructed architecture. By directly passing an `Inventory` object to the `Assets` class, a strong coupling between the classes `Inventory`, `StoreQuery`, and `Assets` is created. Similarly, `Assets`, `Report`, and `Reporting` are coupled strongly due to the other deficiency occurrence. Coupling between classes is a metric that is used by many clustering algorithms to group classes into components. Therefore, it stands to reason that the three components Accounting, Store, and Controlling might be regarded as parts of one larger component by the architecture reconstruction. Thus, it might create a composite component (labeled Accounting/Store/Controlling in Fig. 2) during the reconstruction process containing the three smaller components.

4 The Archimetrix process

In this section, we give an overview of the reengineering process with Archimetrix. The process steps are described in more detail in the subsequent sections. We illustrate the

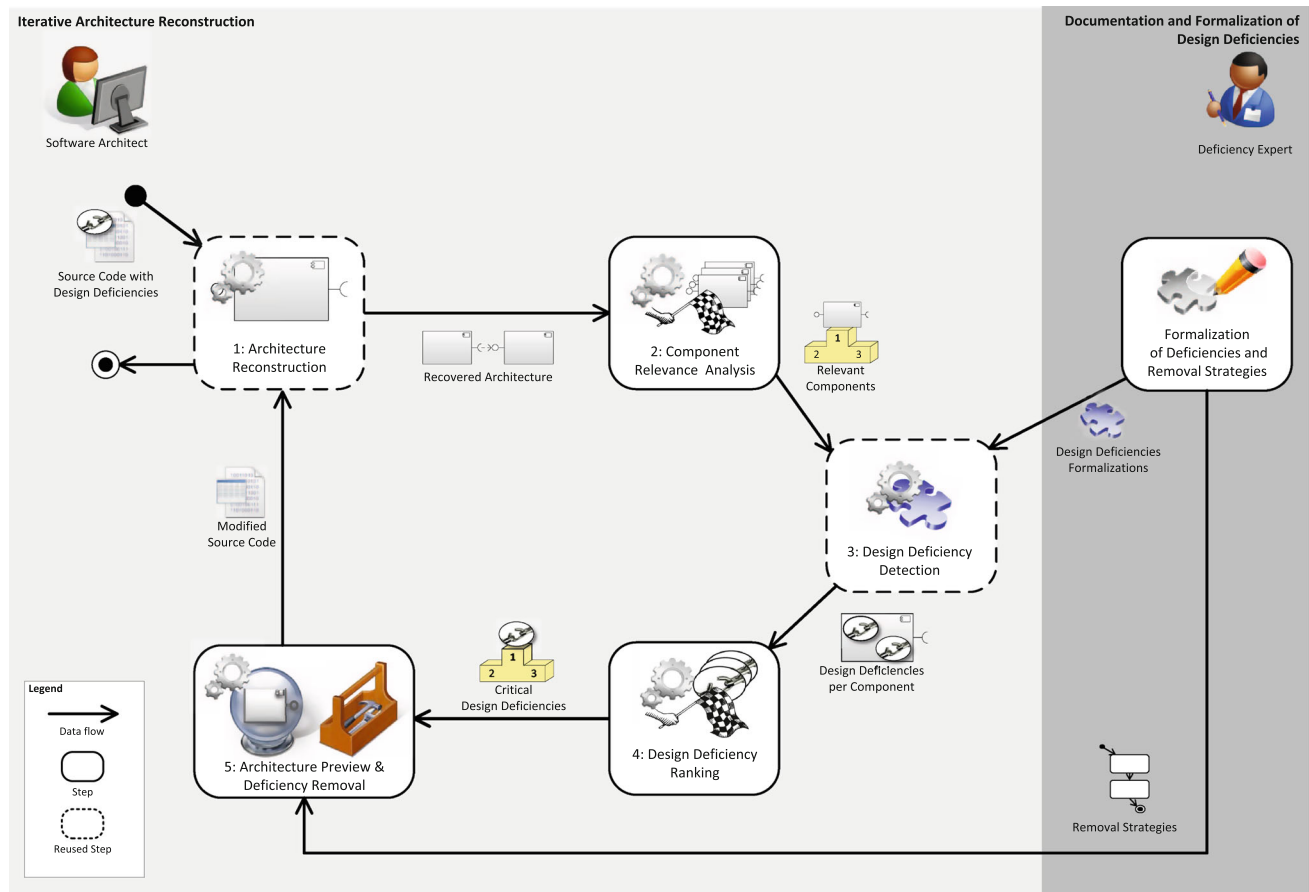


Fig. 3 The Archimatrix process

steps with examples from the store system introduced in the previous section.

Figure 3 shows the Archimatrix process. It consists of two parts: the iterative architecture reconstruction process on the left (shaded in a light gray) and a process for the formalization of design deficiencies (shaded in a darker gray). The former part combines clustering-based architecture reconstruction with design deficiency detection. It is an iterative process with one iteration consisting of five steps. In addition, the deficiencies used in the iterative process have to be discovered, documented, and formalized which is in the focus of the second part. The two parts are executed by different roles: the *software architect* carries out the iterative architecture reconstruction while the *deficiency expert* formalizes the design deficiencies.

The steps in which preexisting approaches are reused are marked with dashed lines in Fig. 3. We do not explain these existing works in detail in this paper but confine ourselves to a description which suffices to understand the Archimatrix process as a whole. In the following, we first explain the iterative architecture reconstruction. Then, the formalization of design deficiencies is described.

4.1 Iterative architecture reconstruction

At the beginning of the process, we assume that the software architect has only the source code of the system. The conceptual architecture is unknown. Also, the architect does not know whether and where design deficiencies exist in the source code.

The process starts with the architecture reconstruction of the system (Step 1). This step does not analyze the source code of the software directly but instead processes a graph representation, the generalized abstract syntax tree (GAST). The GAST represents the syntactic structure of the source code in the form of an abstract syntax graph [2] and contains all references in a resolved form. It is generalized in the sense that it can be used to represent the source code of different languages such as Java, C++, or Delphi. The GAST is created by a parser called SISSy [49].

Based on the GAST, a number of metrics are computed, the metric values are aggregated, and the initial software architecture for the system is reconstructed in accordance with the aggregated values. This initial architecture can provide a first rough overview of the system but it will possibly be

impacted by design deficiencies in the system (see Sect. 3.3). In the example system, the two Transfer Object Ignorance occurrences could lead to an architecture which contains one large component comprising the three conceptual components Accounting, Store, and Controlling. Further details on the architecture reconstruction step are given in Sect. 5.

In order to prevent deficiencies from influencing the reconstructed architecture, we propose to detect and remove the design deficiencies from the system. This has the additional benefit that the software quality improves. However, executing a design deficiency detection on the complete system is very time-consuming [7,48]. As a consequence, we suggest that the software architect should only search for deficiencies in those components in which the search is worthwhile. Ideally, the detection should be focused on components that have a high probability of containing design deficiencies and on components whose reconstruction may have been influenced by deficiencies. To support the architect in the decision which components are a worthwhile input for the design deficiency detection, we added the *Component Relevance Analysis* step (Step 2). It takes the architecture from the initial reconstruction, rates all components, and thereby suggests a sensible input for the design deficiency detection. In our example, the large component Accounting/Store/Controlling could be identified as a viable starting point for the detection of deficiencies. The relevance analysis is explained in detail in Sect. 6.

In the next step, the design deficiency detection (Step 3) can be executed on the selected (relevant) component(s). The architect in our example would arguably choose to search for deficiencies in the large Accounting/Store/Controlling component. The deficiency detection is explained in detail in Sect. 7. It yields a set of design deficiency occurrences in the selected components. For example, at this point, the two Transfer Object Ignorance occurrences depicted in Fig. 2 would be detected.

If a lot of deficiency occurrences are detected, the software architect might not be able to inspect and remove them all, e.g., due to time constraints. Instead, he might want to focus on the most critical deficiencies. However, just by looking at the list of detection results, it is not apparent which deficiencies are more critical than others. To support this decision, Archimatrix performs a *Design Deficiency Ranking* that judges the severity of the detected design deficiency occurrences (Step 4). This ranking mechanism takes all detected design deficiency occurrences as input and ranks them. The two deficiency occurrences in the example could, of course, be compared manually. But for larger numbers of detected occurrences, manual comparisons become more and more tedious and the automatic, heuristic ranking can provide valuable support. Sect. 8 details on the design deficiency ranking.

In order to accomplish the removal of a design deficiency, different removal strategies exist. In some cases, pre-defined removal strategies can be applied automatically. In other,

more complicated cases which are not covered by the strategies, the software architect has to intervene and to remove the design deficiency partly or completely by herself (Step 5).

If a pre-defined removal strategy is applied, its architectural consequences can be visualized by an *Architecture Preview*. This step takes a selected design deficiency occurrence and a chosen removal strategy as input. It produces a comparison of the current architecture and the architecture that would result from the application of the removal strategy. The architect can then preview the effects of different removal strategies on the reconstructed architecture and determine which of the resulting architectures fits her requirements best. If no pre-defined removal strategy can be applied with a satisfying result, the deficiency can also be removed manually.

In our example, occurrence no. 1 of the Transfer Object Ignorance deficiency could be removed by manually creating an appropriate class for the data transfer and using that for the communication between Assets and StoreQuery (see Sect. 3.2). Occurrence no. 2 could be removed by an automated removal strategy which renames the class Report to ReportTO.

The architecture preview and the deficiency removal are described in Sect. 9.

After the software architect has removed one or more deficiency occurrences, the architecture reconstruction can be executed again. The newly reconstructed architecture may differ from the initially recovered one because the removed deficiencies no longer influence the reconstruction. In our simple example, the three components Accounting, Store, and Controlling might no longer be assigned to a common composite component but may instead be recognized as separate components. If the architect is satisfied with the reconstructed architecture, the process ends at this point. Otherwise, the reengineered system is the starting point for a new iteration of the process. Each iteration leads to a reconstructed architecture that contains less deficiencies than the architecture reconstructed by the previous iteration and thus is less influenced by deficiencies.

Note that, in some cases, the removal of one deficiency occurrence can lead to the introduction of other deficiencies. However, as a result of the iterative approach, newly introduced deficiencies will be detected in the next iteration if appropriate formalizations exist. As the whole process is semiautomatic, the software architect can then decide how to act on this newly introduced deficiency. In particular, he will be able to recognize and prevent “infinite loops” (the removal of deficiency A causes deficiency B whose removal in turn causes A).

4.2 Documentation and formalization of design deficiencies

Design deficiencies can only be detected automatically if they are specified in a formal way. However, there is no predeter-

mined, static set of deficiencies for component-based systems in general. While some deficiencies may occur in nearly all systems, others are highly dependent on technology choices or company-specific conventions. Archimatrix comes with four pre-defined design deficiencies which all represent situations in which good component-oriented design is neglected. Archimatrix also allows for the individual documentation and formalization of additional deficiencies. This section describes how deficiencies for a given component-based system can be discovered, documented, and formalized.

Figure 4 shows the process for the documentation and formalization of design deficiencies. Design deficiencies often occur because design principles, guidelines, or conventions are not adhered to. The reasons for this are manifold. For example, high time pressure may force developers to quickly implement a solution without spending much time on devising careful designs. In other cases, inexperienced developers may simply be unaware of existing guidelines or the guidelines may not be sufficiently enforced in a company, e.g., through regular code reviews or version control commit triggers.

The first step in discovering the deficiencies that are to be detected and removed later in the process is to identify principles, guidelines, and conventions that are used in the system under study. These may be common principles of good component-based design like architectural styles or design patterns [3, 24, 51] (Step 1.1). The use of data transfer objects described in Sect. 3 is one example of such a common design principle.

On the other hand, there may be company- or project-specific guidelines which are not universally applicable to arbitrary component-based systems. For example, there may be company-specific naming conventions or development guidelines that arise from the use of a specific framework in the project (Step 1.2). The naming convention for transfer objects presented in Sect. 3 is an example of a project-specific convention. Principles and guidelines like this can be discovered by studying documentation that is in use in the project, e.g., specific textbooks or guideline documents that are used by the developers. On the other hand, developers may also have conventions that are not formally documented and are disseminated by communication among the developers. They can best be learned by interviewing the developers and documenting their knowledge (Step 1.3).

Once the principles, guidelines, and conventions which should have been applied in a system are identified, the deficiencies that arise from their violation can be documented. At first, this can be an informal, textual documentation (Step 2). It is also possible to use pre-defined templates for the documentation of design deficiencies like the AntiPattern templates presented by Brown et al. [13]. The templates can be easily adapted to specific preferences or requirements in a company or project. Sect. 3 shows an example documentation

of the Transfer Object Ignorance deficiency that makes use of Brown's Mini-AntiPattern template. Independent of the formalization and the later detection of deficiencies, this documentation can be very useful for the developers. It can serve as a catalog of bad practices that make developers knowledgeable about possible deficiencies in order to avoid them in the future.

As soon as the deficiencies are identified and documented, they can be formalized (Step 3). In Archimatrix, they are specified as structural object graph patterns in terms of the GAST in a domain-specific, graphical pattern language used by the Reclipse Tool Suite [61–63]. The formalizations represent exemplary situations in which the collected guidelines are *not* adhered to because those are exactly the deficiencies that we want to detect in the system. For example, the principle to exchange data via transfer objects will be represented by a specification that expresses situations which neglect this principle.

Figure 5 displays a formalization of the Transfer Object Ignorance deficiency introduced in Sect. 3. This formalization was created following the process introduced in this section. It describes the exchange of data between classes using a non-transfer object. It is also tailored to the CoCoME system which we used in our validation (see Sect. 11). More precisely, we make use of the convention that transfer objects in CoCoME should be marked by appending the letters "TO" to their name (similar to the convention assumed in the running example). In general, deficiency formalizations can also be created without exploiting such project-specific conventions. However, the incorporation of such specific knowledge may yield more precise detection results. This trade-off between reusability of the specifications and detection precision has to be kept in mind by the deficiency expert.

The specification shows the object structure that constitutes an occurrence of the pattern. In upper left corner of Fig. 5, an object labeled `callingClass : Class` is connected to an ellipse labeled `c1 : Component`. The ellipse represents an annotation of a pattern that has been created earlier in the pattern detection process. In this case, it marks the component `c1` to which the `callingClass` belongs. The calling class contains a `MethodCall` to a target called `Method`. The `calledMethod` is contained in a `calledClass` which belongs to a component `c2`. The `calledMethod` has a parameter `param` whose type `paramType` also belongs to component `c1`. The `paramType` is not marked as a transfer object which is signified by the constraint that is imposed on its name. It states that the name may be arbitrary but may not end with the suffix "TO." This expression uses the RegEx language for regular expressions from Java.

Together with the deficiencies, the deficiency expert can also create automated removal strategies. In Archimatrix, these strategies are formalized as model transformations of the GAST. We reuse the graphical, in-place model transformation language *Story Diagrams* for this purpose [60].

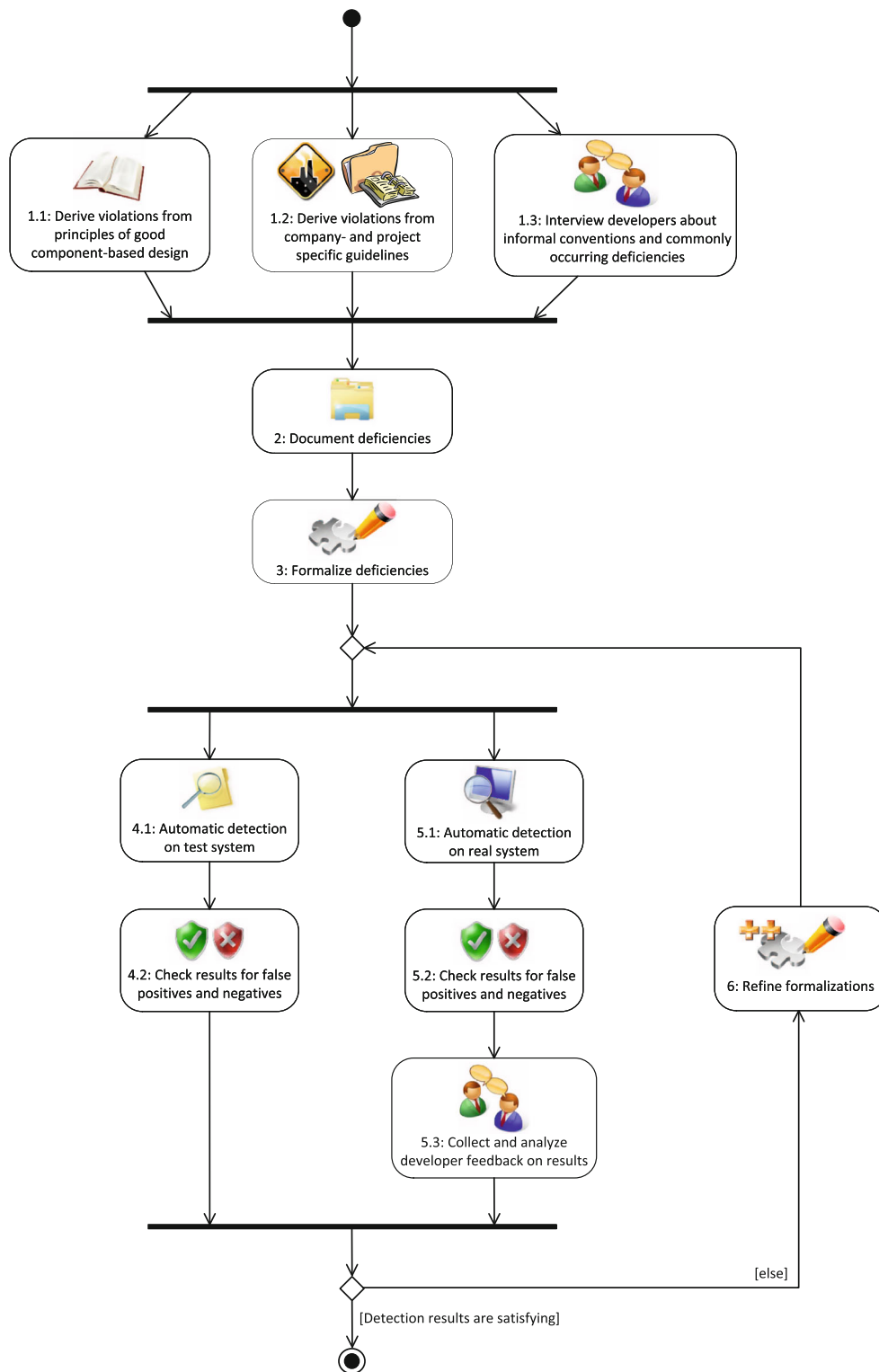


Fig. 4 Process for the documentation and formalization of design deficiencies

Figure 6 shows a simplified, illustrative example of a removal strategy. It creates a method declaration in an interface and redirects a call from a previously called method to this new declaration. This is a central part of the first removal strat-

egy for the Transfer Object Ignorance deficiency described in Sect. 3.2.

In the example, the strategy's name and its parameters are shown on the top. Two parameters are passed to the removal

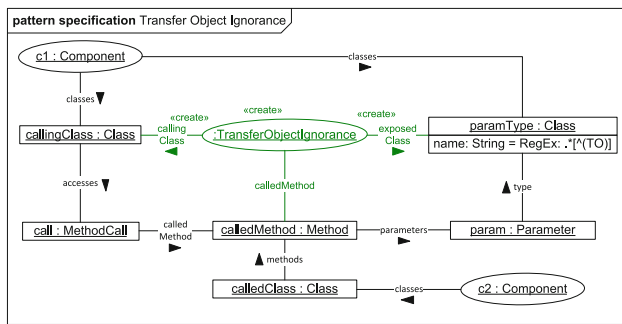


Fig. 5 Formalization of the transfer object ignorance pattern

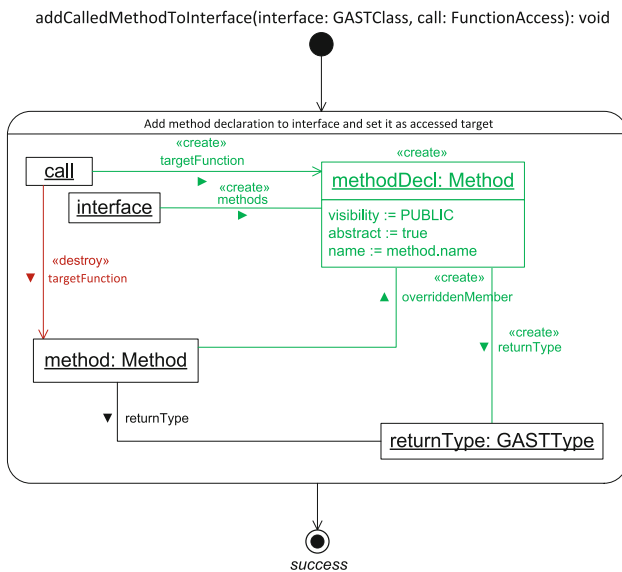


Fig. 6 Formalization of a removal strategy

strategy: the interface which will be extended and the call to the method which is to be added to the interface. Similar to the deficiency formalization, black elements represent objects and links which already exist in the GAST and have to be matched. Red elements marked with `«destroy»` are deleted by the strategy while green elements marked with `«create»` are created. In the example, the old target of the call was the method which is shown to the left of the diagram. A new method `methodDecl` is created and added to the interface (represented by the connection between interface and `methodDecl`). The connection between the call and the old method is destroyed and the `methodDecl` is set as the new target of the call. In addition, the `methodDecl` is set to be overridden by the method. The `methodDecl` is set to have a public visibility and to be abstract. Its name is set to the name of the old method. The `returnType` is also set to be identical. This simple example is only meant to provide an idea of the automated removal strategies and leaves many things unconsidered, e.g., the parameters of `method` and `methodDecl`. In reality, the strategies can become much more complex.

The formalization of deficiencies is a non-trivial task that has to be carried out manually. Therefore, a validation is necessary to determine whether the formalizations are suited to detect the deficiencies they are meant to detect. This is accomplished in two different ways. On the one hand, a search for the specified deficiencies can be executed on a test system (Step 4.1). The test system usually is a small, manually created system which deliberately contains occurrences of the deficiencies in question. The detection results are then analyzed for false positives and false negatives (Step 4.2). If there are false positives, i.e., detected deficiency occurrences that resemble the specifications structurally but are not really deficiencies, the specifications are not sufficiently exact. A false negative is a deficiency occurrence that was implemented in the test system but that is not detected. In this case, the specification may be too restrictive.

Another possibility to validate the deficiency specifications is to execute a deficiency detection on a part of a real system (Step 5.1). Ideally, the specified deficiencies are already known to occur in that part. After the detection, the detection results can be analyzed for false positives and negatives (Step 5.2). Of course, the false negatives can only be determined with respect to the known deficiency occurrences. Afterward, the detection results should be discussed with the developers that are responsible for the analyzed part of the system, provided they are still available (Step 5.3). They can give additional insight into the occurrence of the deficiencies and can also judge whether the detection results are of interest to them.

If the validation shows that the detection results are satisfying, the formalization of the deficiencies ends. Otherwise, the formalizations can be adapted in accordance with the obtained insight (Step 6). Afterward, the validation (Steps 4.1–5.3) and adaptation can be repeated until a sufficiently good detection result is achieved.

The formalizations of all the deficiencies used in the validation are available on the Archimatrix Web site [1]. A more detailed explanation of the pattern formalization language and more example patterns can be found in [41, 54, 61–63].

5 Architecture reconstruction

For the architecture reconstruction (Step 1 in Fig. 3), we reuse the Software Model Extractor (SoMoX) which was developed at the Karlsruhe Institute of Technology [8, 16, 33]. SoMoX is an architecture reconstruction approach that uses clustering to infer a software architecture from source code. As we use SoMoX as a black-box component in our process, we do not explain it in detail in this paper. We rather give an idea of how SoMoX works to allow for the understanding of the subsequent process steps.

SoMoX computes a number of clustering metrics such as name resemblance, coupling, and cohesion for the elements of the GAST. The inputs for SoMoX are the GAST, user-configurable weights for the clustering metrics, and the thresholds for the merging and composition of component candidates (explained below). SoMoX creates two outputs: First, a formal model of the system architecture in form of a number of components and their connections. Second, a mapping of all source code elements (i.e., classes and interfaces) to the components in the architecture.

For illustrative purposes, we explain one example of a clustering metric here: the *Package Mapping* metric. Package mapping expresses the distance between two classes in terms of their position in the package tree of Java systems. The rationale behind this clustering metric is that packages are often organized such that they contain related classes and are therefore a good indicator for the reconstruction of components. The formula for the package mapping metric is given in Eq. 1.

$$PM(A, B) = \frac{\text{commonRootHeight}(A, B)}{\text{maxHeight}(A, B)} \quad (1)$$

Here, *maxHeight* designates the maximum height of the package tree for the classes *A* and *B*. The *commonRootHeight* represents the height of the deepest common node in the package tree of those classes [33]. Classes that lie in the same package receive a package mapping value of 1, classes in entirely different parts of the system receive a package mapping of 0.

SoMoX works iteratively. Each iteration builds on the results of previous iterations and tries to create new components from the components reconstructed so far. (These clustering iterations of SoMoX are not to be confused with the iterations of the Archimatrix process.) The initial iteration produces one primitive component for each class. In the subsequent iterations, every combination of two components, either primitive or composite, constitutes a component candidate, i.e., a candidate for the creation of a new component by combining two smaller ones. There are three possibilities for combining a component candidate (see Fig. 7): (a) If the candidate consists of two primitive components, it can be merged, i.e., unified into one larger primitive component. (b) A candidate can also be composed, i.e., encapsulated in a new composite component. This can happen for both, primitive and composite, components. (c) Finally, a candidate can also be discarded. In this case, no new component is created.

To determine whether a component candidate is merged, composed, or discarded, its clustering metric values are calculated and aggregated into a merging value v_{merge} and a composition value v_{compose} for every potential candidate. For example, v_{merge} is computed by calculating the weighted sum of those clustering metrics that measure the bypassing of interfaces, the name resemblance, the balance of abstract

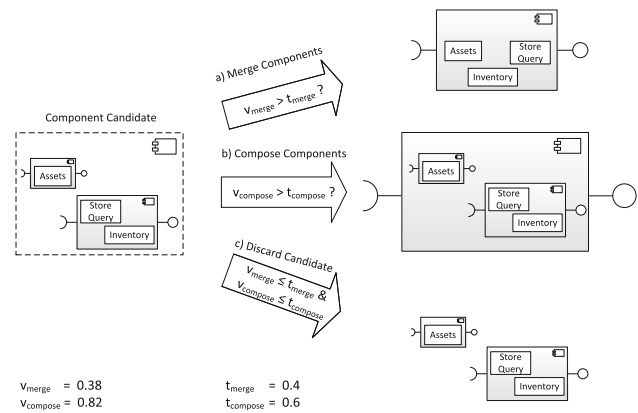


Fig. 7 Merging and composition of components in SoMoX

and concrete classes, and the package mapping of a candidate. As we just reuse SoMoX, the detailed explanation of these clustering metrics is outside the scope of this paper.

If the merging value for a candidate exceeds a certain merging threshold t_{merge} , the candidate is merged. If the composition value exceeds a composition threshold t_{compose} , the two components that constitute the candidate are composed. If both thresholds are exceeded, merging takes precedence over composition. If both aggregated values are below the respective thresholds, the candidate is discarded. In the example in Fig. 7, v_{merge} is 0.38 and therefore below the threshold t_{merge} of 0.4. In contrast, v_{compose} is 0.82 and therefore exceeds the threshold of 0.6. Hence, the candidate would be converted into a new composite component. t_{merge} and t_{compose} are configured by the user before the clustering begins. After each iteration, t_{merge} is increased while t_{compose} is decreased. This way, component merges are more likely at the beginning of the clustering while component compositions become more likely toward the end. Then, new candidates are constructed from the recovered components and the next iteration commences. The clustering process ends when no new components can be produced by merging or composing component candidates during one iteration.

6 Component relevance analysis

After the architecture has been reconstructed, the software architect has to decide which components should be searched for design deficiencies, i.e., which components are most *relevant* for the deficiency detection. To support this decision, Archimatrix provides the *component relevance analysis*, a heuristic which identifies these relevant components (Step 2 in Fig. 3). To rate a component's relevance, we currently use two different *relevance metrics*: the *Complexity Metric* and the *Closeness to Threshold (CTT) Metric*. For both relevance metrics, the range is $[0 \dots 1]$ and the metric values are pro-

portional to the relevance of the component in question, i.e., the higher the metric value, the more relevant the component. This section explains the rationale and the calculation of these metrics and illustrates how a component's relevance can be determined. In the future, this approach can be easily extended by adding more relevance metrics.

6.1 Complexity metric

The *Complexity Metric* identifies complex components. Studies show that components consisting of many classes, methods, and interfaces are hard to understand and are therefore difficult to maintain and to adapt [11,26,44]. This situation worsens over time. Thus, the risk of inadvertently introducing design deficiencies increases. This leads to the following assumption: the more complex a component, the more likely it is to contain design deficiencies. This makes the complexity of a component a significant indicator of its relevance.

To calculate the complexity, we use the component plain complexity (CPC) metric as described by Cho et al. [15]. It is expressed by the following formula:

$$\begin{aligned} CPC(C) = & \#Classes(C) + \#Interfaces(C) \\ & + \sum_{c \in Classes(C)} \#Methods(c) \\ & + \sum_{c \in Classes(C)} \#Attributes(c) \\ & + \sum_{c \in Classes(C)} \sum_{m \in Methods(c)} \#Params(m) \end{aligned} \quad (2)$$

In Formula 2, $CPC(C)$ is a sum of the number of classes, interfaces, methods, attributes, and method parameters in the component C . Therefore, the component plain complexity is the sum of the component's size (in terms of contained classes) and the complexity of its elements.

In order to obtain a relevance metric value between 0 and 1, we normalize the value by relating it to the complexity of the most complex component (see Eq. 3).

$$CPC(C)_{norm} = \frac{CPC(C)}{\max_{C_i \in allComponents} (CPC(C_i))} \quad (3)$$

Note that the component plain complexity is a very basic heuristic for the calculation of the component complexity. In the future, the complexity metric could be refined by taking metrics like McCabe's cyclomatic complexity [37] or the design evolution metrics proposed by Kpodjedo et al. [30] into account.

6.2 Closeness to threshold metric

During the clustering step, components are merged or composed based on their aggregated metric values v_{merge} and

$v_{compose}$. We showed in our previous work that the presence of design deficiencies in the code can influence the metric values and therefore the clustering results [58,59]. The *CTT Metric* introduced in this section identifies components for whom one of the aggregated metric values was close to the merging or composition threshold, i.e., the metric value was just high enough to merge or compose the component candidate (or just low enough to discard it, respectively).

If a design deficiency in such a component is detected and removed, this will change v_{merge} and $v_{compose}$ and possibly send one of them across the threshold. This will lead to a different clustering result, i.e., to a different reconstructed architecture.

If a component was created through merging, its distance from the threshold ε is calculated as shown in Formula 4. If it was created through composition, the distance is calculated accordingly (see Formula 5).

$$\varepsilon = |t_{merge} - v_{merge}| \quad (4)$$

$$\varepsilon = |t_{compose} - v_{compose}| \quad (5)$$

The closeness to the threshold is then determined by subtracting ε from 1 (see Formula 6).

$$CTT(C) = 1 - \varepsilon \quad (6)$$

Components with an aggregated metric value close to the threshold receive a high CTT value and are therefore rated as very relevant. The farther a component's aggregated metric values are from the threshold, the less relevant it is ranked by the CTT metric. Thus, the CTT metric allows the software architect to focus on design deficiencies whose removal will probably have an impact on the system's architecture.

6.3 Relevance calculation

Both relevance metrics are normalized to a value between zero and one. In the following heuristic, we assume that all relevance metrics are equally important. To identify the most relevant components, the Pareto optimality [17] is calculated. A Pareto optimal set contains solutions that represent the best possible trade-off among a number of objectives. A solution is called Pareto optimal if and only if there is no solution that dominates it. Here, we use the *dominates* relation in a maximization context: A solution y dominates a solution z iff $\forall i \in [1 \dots n], f_i(y) \geq f_i(z)$ and $\exists i \in [1 \dots n]$ such that $f_i(y) > f_i(z)$. In the component relevance analysis, each relevance metric (e.g., the ones presented in Sects. 6.1 and 6.2) is used as an objective function of the Pareto optimal search. The components that are Pareto optimal with respect to all the relevance metrics are assumed to be good subjects for a design deficiency detection because they represent the best available combination of relevance indicators.

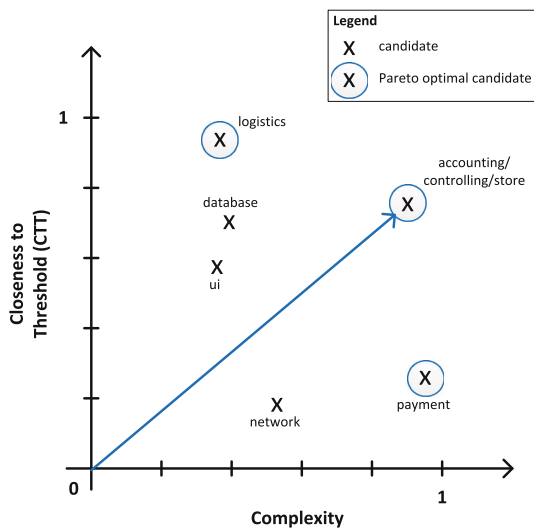


Fig. 8 Relevance analysis result calculation

Figure 8 illustrates exemplary results of the component relevance analysis for our running example. The relevance values are visualized in a coordinate system. The x-axis represents the component complexity (see Sect. 6.1) and the y-axis represents the closeness to threshold (see Sect. 6.2). The six components from the example system are positioned in the coordinate system. The component accounting/controlling/store represents the combination of the three individual components. The Pareto optimal components accounting/controlling/store, logistics, and payment are marked with circles. They build up a Pareto front. Each candidate below the Pareto front is dominated by the other candidates; hence, it is not Pareto optimal (here: database, ui, and network) and therefore less interesting.

If several Pareto optimal solutions exist, as in the example in Fig. 8, a further criterion to identify the most relevant component is required. For this purpose, the geometric distance to the origin is used as a heuristic: The higher this distance, the more relevant the corresponding component. Note that this is possible due to the proportionality between the metric values and the component relevance.

In our example, accounting/controlling/store is the component with the largest distance (indicated by the arrow) and thus it is the most relevant for the design deficiency detection.

This approach to calculate an overall relevance value is easily extensible. The Pareto optimality as well as the geometric distance are computable for arbitrarily many dimensions. Thus, any number of new relevance metrics can be added to rate the relevance of a component given their metric are proportional to the component relevance. As the software architect can examine both, the aggregated relevance value as well as the values of the individual metrics, he can also base his decision which component to search for deficiencies on a personal selection of relevance metrics, thereby ignoring, e.g., the complexity of components.

7 Design deficiency detection

After the component relevance analysis, the detection of deficiencies can begin (Step 3 in Fig. 3). The deficiencies have to be formalized as described in Sect. 4.2. The deficiency detection is accomplished by using the Reclipse Tool Suite that was developed by our group in earlier work [61–63].

Reclipse employs a graph matching approach for the detection of structural patterns [40]. The GAST of the software system is the host graph for this matching. Reclipse detects occurrences of a specified graph pattern (e.g., the formalization of the Transfer Object Ignorance deficiency in Fig. 5) by searching an isomorphic matching between the specified object graph pattern and the host graph. When a pattern occurrence in the host graph is detected, Reclipse creates an annotation (e.g., the green ellipse labeled with TransferObjectIgnorance in Fig. 5) which marks the pattern occurrence.

In general, subgraph matching is NP-complete [20]. In order to allow for an efficient detection of deficiency occurrences, we limit the search scope of the detection. The component relevance analysis (see Sect. 6) supports the architect in determining which components are relevant for the pattern detection. It enables the architect to choose a good subset of components for the deficiency detection and allows for a faster detection.

8 Design deficiency ranking

The design deficiency detection results in a list of detected deficiency occurrences. However, the software architect may not be able to inspect and remove them all, e.g., because of time constraints. To support the architect in the decision which of these occurrences should be removed, the occurrences are ranked based on their influence on the reconstructed architecture (Step 4 in Fig. 3).

Similar to the component relevance analysis, the design deficiency ranking uses a number of ranking metrics to rank a design deficiency occurrence according to its severity. The applicability of a ranking metric to a design deficiency occurrence depends on the design deficiency. For the Transfer Object Ignorance design deficiency, the heuristic ranking metrics *Class Locations* and *Communication via Data Classes* are applicable. We explain these strategies and their combination in the remainder of this section.

8.1 Class locations metric

The idea behind the *Class Locations Metric* is that classes that reside in the same part of the system are intended to collaborate closely with each other. For Java systems, these are classes that lie in the same branch of the pack-

age tree or even belong to the same package. Consequently, an occurrence of the Transfer Object Ignorance deficiency between classes that are located far away from each other (in terms of the package structure) is a more serious design problem than an occurrence between classes in the same part of the package tree. For this metric, the value of the *PackageMapping* (PM) metric that is calculated during the architecture reconstruction is reused (see Formula 1 in Sect. 5).

The calculation of the class locations metric is shown in Formula 7.

$$Rank_{CL}(D) = 1 - PM(CC_D) \quad (7)$$

Here, D is the design deficiency occurrence and CC_D represents the component candidate that contains the classes that are involved in the design deficiency occurrence D . In the running example, the involved classes would be matches of the calledClass, the callingClass, and the paramType from the formalization in Fig. 5. The higher the *PackageMapping* value, the lower the occurrence is ranked.

In our running example, the classes Assets and Report lie in the package app.accounting while the class Reporting belongs to the package app.controlling (see Fig. 2). Because they belong to different sub-packages of the app package, they have a package mapping value of 0.5. Thus, the rank of the Transfer Object Ignorance deficiency no. 2 would be $1 - 0.5 = 0.5$.

Deficiency occurrence no. 1 in the example exists between the classes Assets, StoreQuery, and Inventory. They lie in the packages app.accounting and data.storage, respectively. Thus, the classes have a package mapping value of 0 and the deficiency's rank is 1, i.e., it is more relevant than deficiency no. 2.

8.2 Communication via data classes metric

The Transfer Object Ignorance deficiency describes situations in which objects that are not transfer objects are passed between components. There are several degrees of severity depending on the class that is used for communication. For example, a class that only contains fields and access methods but does not adhere to the specific naming convention for transfer objects does not constitute a major problem. It may be that the developer intended to use a transfer object and only forgot to name the class correctly. If, on the other hand, a class adheres to the naming convention but contains several methods with application logic, the deficiency is more severe. Hence, one possible heuristic is this: The more non-accessor methods occur in the class, the worse is the deficiency. Therefore, we use the *Communication via Data Classes Metric* (see Formula 8).

$$Rank_{DC}(D) = 1 - IsDataClass(c) \quad (8)$$

$$IsDataClass(c) = \begin{cases} 0 & \text{if } \#Fields(c) = 0 \\ 1 - \left(\frac{\#NonAcc.(c) + \#MissingAcc.(c)}{\#NonAcc.(c) + \#PotentialAcc.(c)} \right) & \text{else} \end{cases} \quad (9)$$

The metric value of the communication via data classes metric for Transfer Object Ignorance occurrences is 1 minus the similarity of the transfer object's class to a data class. Formula 9 shows how this similarity is calculated. If the class does not contain any fields, it is definitely not a data class. Therefore, the rank of such a deficiency occurrence with respect to the communication via data classes metric is 1. If the class contains fields, the metric value depends on the methods in the class. On the one hand, the number of methods that are not accessors, $\#NonAcc.(c)$, is counted. The number of potential accessor methods in a class, $\#PotentialAcc(c)$, is 2 times the number of fields. By counting the number of actual accessor methods and subtracting the count from the number of potential accessor methods, we get the number of missing accessor methods, $\#MissingAcc.(c)$. The method counts are related to one another as shown in the formula. The more non-accessor methods exist in the class and the more accessor methods are missing, the more the class deviates from a data class. This leads to a higher ranking with respect to this metric.

In Sect. 3, we state that the class Inventory is exposed to the Accounting component, although it is not a data transfer object. Inventory has one non-accessor method: checkStock. Assuming that the Inventory class possesses two access methods for the field items (e.g., getItem and setItem), no accessor methods are missing. The rank calculation for the Transfer Object Ignorance occurrence between Assets, StoreQuery, and Inventory would be as follows:

$$IsDataClass(Inventory) = 1 - \left(\frac{1 + 0}{1 + 2} \right) = \frac{2}{3}$$

$$Rank_{DC}(TOI_1) = 1 - \frac{2}{3} = \frac{1}{3}$$

In contrast, Report does not contain non-accessor methods. Therefore, the Transfer Object Ignorance occurrence between Assets, Reporting, and Report would be ranked like this:

$$IsDataClass(Report) = 1 - \left(\frac{0 + 0}{0 + 2} \right) = 1$$

$$Rank_{DC}(TOI_2) = 1 - 1 = 0$$

8.3 Rank calculation

The overall heuristic ranking result for the design deficiency occurrences is determined by calculating the Pareto optimal design deficiencies with respect to the applicable ranking metrics. This is similar to the calculation of the component

relevance described in Sect. 6.3. Thus, the design deficiency ranking is also easily extensible. Again, the software architect could also choose to examine the individual ranking metric values instead of relying on the aggregated rank. Thereby, he can focus on the ranking metrics that interest him the most.

For other design deficiencies, different sets of metrics are applied. For example, for the design deficiency *Interface Violation* [59], the metrics *Class Locations*, *Number of External Accesses*, and *High Interface Adherence* are used. These metrics are explained in detail in [41, 58].

9 Architecture preview and deficiency removal

This section discusses how Archimetrix supports the removal of detected deficiency occurrences. We also present the architecture preview which visualizes the consequences of the application of automatic removal strategies (Step 5 in Fig. 3).

As explained in Sect. 4, a number of different removal strategies may be conceived of for each deficiency. For the example in Sect. 3, we described three different possibilities for the removal of Transfer Object Ignorance deficiencies. Detected deficiencies can be removed manually or by applying automatic removal strategies.

9.1 Automatic deficiency removal

Sometimes, the removal of a deficiency is straightforward and can be automated. The automatic removal strategies can be implemented in different ways. Since Archimetrix operates on the GAST of the system, model transformations are a possible way to reengineer the system.

In Archimetrix, we use the in-place model transformation language *Story Diagrams* [22] to specify the removal strategies. When applied to remove a specific deficiency occurrence, a story diagram transforms the GAST of the software. The transformed GAST can then serve as the input for the next clustering step.

In order to also manifest the automatic removal in the code, the software architect has to either change the code manually, or the transformed GAST can be used to generate new code. (The latter is possible since all the relevant information from the code and also trace links to the code are retained in the parsing step.) However, if the intention of the architect is mainly to get a precise architectural view of the system, it may not even be necessary to actually remove the deficiencies from the code. In that case, the reconstructed architecture would serve as a pure analysis model.

9.2 Manual deficiency removal

In simple cases, using an automatic strategy is quicker and more convenient than a manual reengineering. However, there may be deficiency occurrences that are built into the

system in complicated or unexpected ways that were not foreseen by the developers of the removal strategies. Others may require human judgment to determine whether they really need to be removed. Thus, it is not always possible to use an automatic removal strategy for the removal of a deficiency. In those cases, removing the deficiency occurrence from the code manually is inevitable.

9.3 Architecture preview

To support the software architect in the decision which removal strategy is most suited to remove a given design deficiency, the architectural consequences of the automatic removal strategies can be previewed. For that purpose, the architecture that will result from the automatic removal of a given design deficiency is calculated and presented to the architect. This preview helps the architect in judging how the selected removal strategy affects the architecture and lets her decide whether this is in line with her expectations. Note that the architecture preview is only available for automatic removal strategies. If the deficiencies are removed manually, no preview can be presented. Archimetrix does not provide support for previewing and undoing the effects of a manual deficiency removal at the moment.

In the architecture preview, the architecture resulting from the initial clustering (referred to as *original architecture*) is compared to the anticipated architecture from the preview (referred to as *previewed architecture*). To execute the architecture preview, the software architect selects the design deficiency occurrence to be removed and a removal strategy to accomplish the removal. The selected removal strategy is applied to a copy of the original architecture by executing an in-place model transformation. The previewed architecture is then calculated by executing a new clustering on the modified copy of the model. As the clustering scales well for large systems (the developers of SoMoX report clustering times between 4 and 12 min for a system with 250,000 LOC [29], depending on the configuration), the execution of a complete clustering for producing the previewed architecture is justifiable.

To simplify the comparison for the user, the differences between the original architecture and the previewed architecture can be visualized. Figure 9 depicts a possible visualization of the architecture preview for our running example. In this figure, the components of the original architecture are visualized on the left side and the components of the previewed architecture are visualized on the right side. In this example, the original architecture only consists of one component named Accounting/Store/Controlling. The previewed architecture consists of three components, Accounting, Store, and Controlling. Moved classes are visualized with a darker background in this figure. The reason for the new component structure is that the classes StoreQuery, Inven-

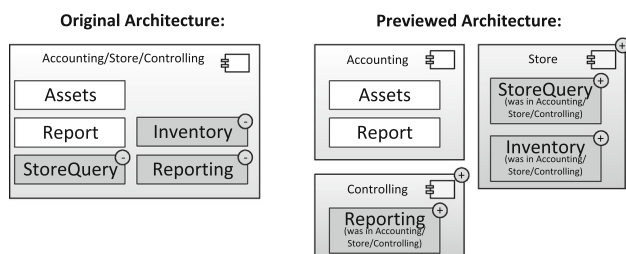


Fig. 9 Architecture preview example

tory, and Reporting are assigned to different components after the removal of the design deficiencies. In the figure, this is marked in the original architecture with a minus sign on the classes. Elements that are new in the previewed architecture in comparison with the original architecture are marked with a plus. In this case, this applies to the components Store and Controlling and their classes StoreQuery, Inventory, and Reporting. For these classes, in the previewed architecture, the label “was in Accounting/Store/Controlling” indicates their former location.

10 Tool support

In order to support the Archimetrix process, a research prototype has been implemented as a plug-in for eclipse. It is freely available for download [1]. The current implementation allows the analysis of Java, C++, and Delphi code (of which only Java has been tested in-depth).

Archimetrix reuses existing tools for several of the process steps described in Sect. 4. Parsing the source code into a GAST is accomplished by a parser named SISSy [49]. For the architecture reconstruction, Archimetrix relies on a tool called SoMoX [16] that uses a combination of software metrics to heuristically create a model of the system’s components and their connectors. The deficiency detection is accomplished by Reclipse [61], a pattern detection tool which employs graph matching to identify patterns in an GAST. The automated removal of detected deficiency occurrences is realized by transformations of the GAST. For this step, we use interpreted story diagrams [60], a graphical in-place model transformation language. The remaining steps of the process (component relevance analysis, deficiency ranking, and architecture preview) are implemented within Archimetrix itself. Archimetrix is also responsible for the overall coordination and execution of the process steps.

Figure 10 shows a snapshot of an architecture model that was reconstructed by SoMoX. It shows a number of reconstructed components, their interfaces, and connectors. For every composite component, a similar diagram can be opened which shows the components contained therein.

Figure 11 shows a snapshot of the editor for the formalization of deficiencies offered by Reclipse. In the figure, a

formalization of the Interface Violation deficiency can be seen.

The snapshot in Fig. 12 shows a part of the relevant deficiencies view. It lists two different detected deficiency occurrences and the corresponding metric values as described in Sect. 8. The occurrences are highlighted in yellow because they are Pareto optimal with respect to their relevance.

The architecture preview of our prototype is shown in Fig. 13. In contrast to the preview sketched in Fig. 9, the tool does not yet support a graphical comparison at the moment. However, it shows a number of relevant values such as the number of primitive and composite components for the original and the previewed architecture. It also shows the two architectures side by side in a tree view (see the lower part of Fig. 13). On the left, the components of the original architecture are shown. The components of the previewed architecture are shown on the right. The number of contained classes is shown behind every component. Composite components can be expanded to reveal the contained components and classes. In addition, components that are no longer present in the previewed architecture are highlighted in red. Components which have changed are highlighted in yellow. New components would be marked in green in the previewed architecture (not shown in the figure). The implementation of a graphical comparison view is future work.

11 Validation

This section describes the validation of Archimetrix. First, we outline our evaluation goals by introducing research questions and the assumptions they are based on. After that, we describe which systems we chose as evaluation subjects and how the defined research questions can be answered based on this selection.

We evaluated Archimetrix with respect to the following research questions:

- Q1. Do the design deficiencies defined by us occur in practice in component-based system implementations even if the systems under analysis were developed in a strictly component-based way?
- Q2. Are our deficiency formalizations sufficiently precise to detect actual deficiencies instead of false positives, i.e., can we detect deficiencies with a high precision? And are the formalizations sufficiently general to detect all occurrences of the respective deficiencies, i.e., do we achieve a high recall? (This question corresponds to Step 5.2 in Fig. 4.)
- Q3. Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile?

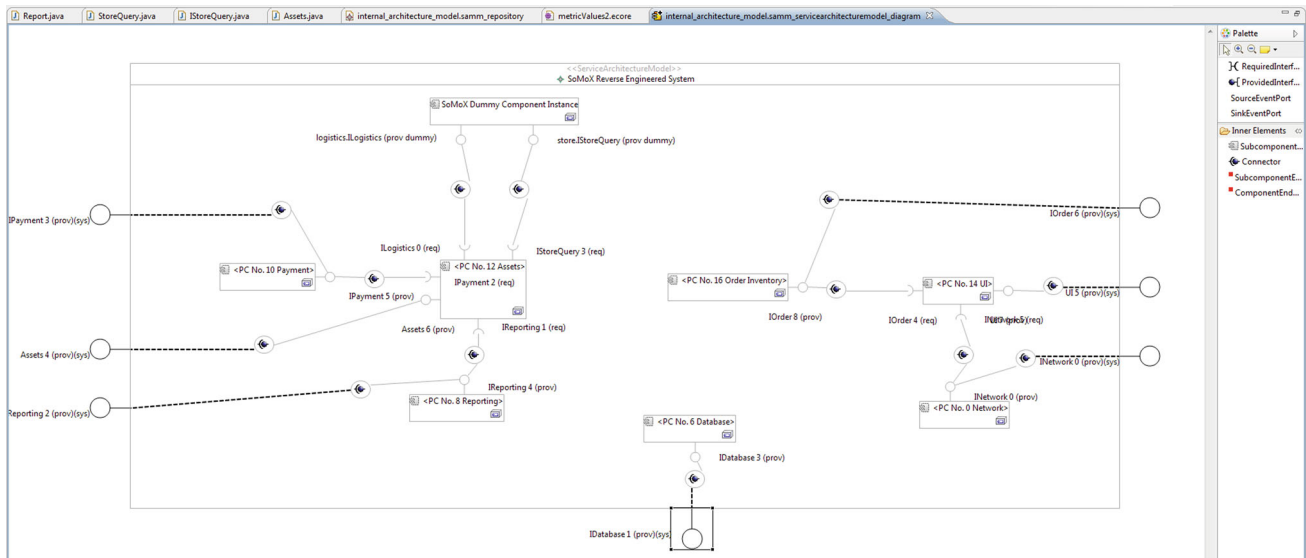


Fig. 10 Snapshot of a reconstructed architecture model

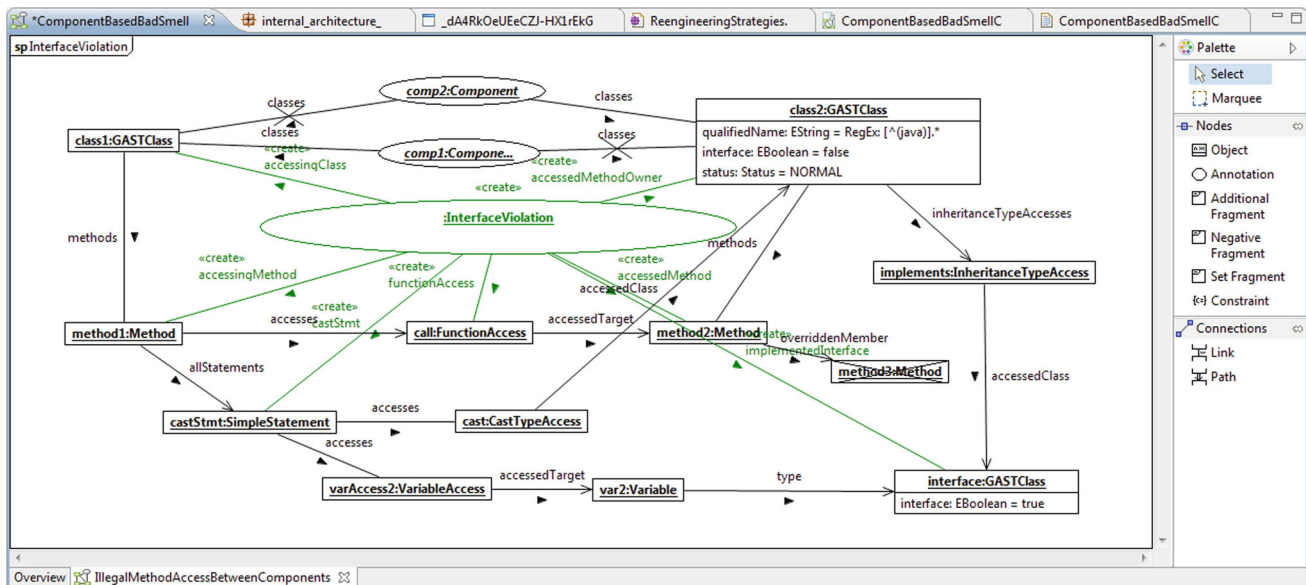
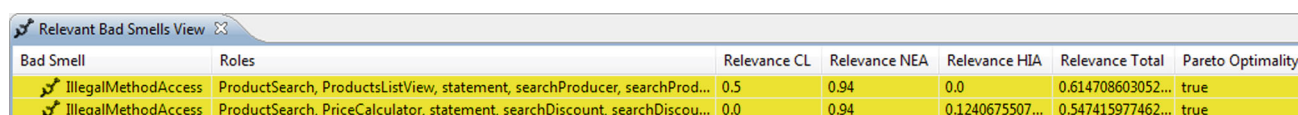


Fig. 11 Snapshot of a deficiency formalization

- Q4. How does the limitation of the scope for the deficiency detection by the relevance analysis improve scalability with respect to pure pattern matching?
- Q5. Does the removal of the deficiencies that receive a high ranking value lead to architectural changes and does the removal of deficiencies with a low ranking value leave the architecture unchanged, i.e., do the deficiency ranking heuristics work?
- Q6. Is the reconstructed architecture after the removal of a relevant design deficiency closer to the documented architecture?

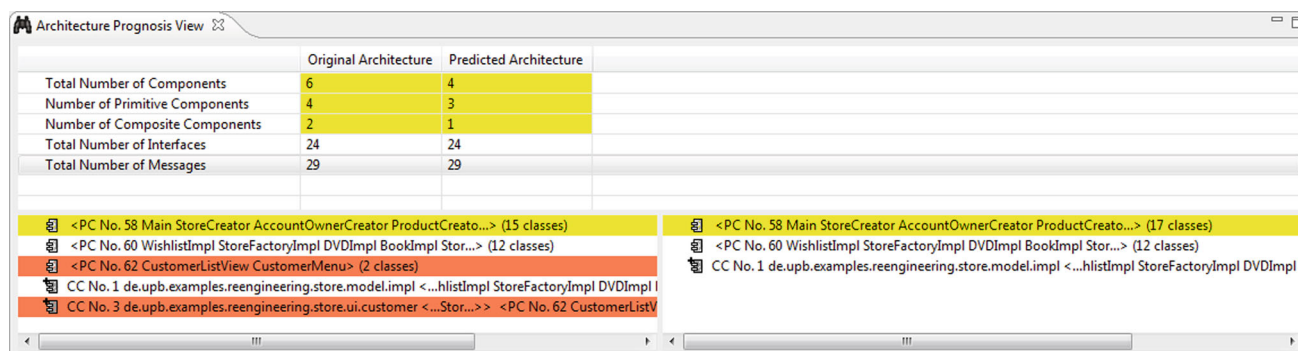
To answer these questions, we selected two case studies in which we applied the Archimatrix process to three different code bases of two different systems.

First, we applied our approach to Palladio Fileshare [34]. Palladio Fileshare has been chosen because we assume it not to contain many design deficiencies as it is a system that has been designed intentionally in a strictly component-based way. Thereby, it is suitable to validate whether the application of Archimatrix produces false positives. Hence, with our results of the application on Palladio Fileshare, we target research questions Q1 and Q2.



Bad Smell	Roles	Relevance CL	Relevance NEA	Relevance HIA	Relevance Total	Pareto Optimality
IllegalMethodAccess	ProductSearch, ProductsListView, statement, searchProducer, searchProd...	0.5	0.94	0.0	0.614708603052...	true
IllegalMethodAccess	ProductSearch, PriceCalculator, statement, searchDiscount, searchDiscou...	0.0	0.94	0.1240675507...	0.547415977462...	true

Fig. 12 Snapshot of a view showing the relevant deficiency occurrences



	Original Architecture	Predicted Architecture
Total Number of Components	6	4
Number of Primitive Components	4	3
Number of Composite Components	2	1
Total Number of Interfaces	24	24
Total Number of Messages	29	29

Original Architecture	Predicted Architecture
<PC No. 58 Main StoreCreator AccountOwnerCreator ProductCreato...> (15 classes)	<PC No. 58 Main StoreCreator AccountOwnerCreator ProductCreato...> (17 classes)
<PC No. 60 WishlistImpl StoreFactoryImpl DVDImpl BookImpl Stor...> (12 classes)	<PC No. 60 WishlistImpl StoreFactoryImpl DVDImpl BookImpl Stor...> (12 classes)
<PC No. 62 CustomerListView CustomerMenu> (2 classes)	CC No. 1 de.upb.examples.reengineering.store.model.impl <...hlistImpl StoreFactoryImpl DVDImpl I
CC No. 1 de.upb.examples.reengineering.store.model.impl <...hlistImpl StoreFactoryImpl DVDImpl I	
CC No. 3 de.upb.examples.reengineering.store.ui.customer <...Stor...> <PC No. 62 CustomerListV	

Fig. 13 Snapshot of the architecture preview view

Additionally, we analyzed and compared the results for the Common Component Modeling Example (CoCoME) [43]. CoCoME was created as a benchmark system. Thus, it allows us to generalize our results. Furthermore, as its conceptual as well as its concrete architecture are available, it becomes possible to evaluate the results of the validation of Archimatrix (although the Archimatrix process in general assumes that no architecture documentation is available). In addition to a reference implementation, there are a number of other CoCoME implementations which make use of different component frameworks like SOFA or FRACTAL [43]. These frameworks are intended to support the creation of component-oriented systems, e.g., by prohibiting the external access to methods which are not part of a component's interface. Thus, we applied Archimatrix to the reference implementation and to the SOFA implementation of CoCoME.

We expect the SOFA implementation of CoCoME to represent a good design since a well-established component framework enforcing good design has been used. In contrast, the reference implementation of CoCoME has been implemented by students in plain Java and therefore is expected to contain more deficiencies on which the relevance analysis, the deficiency ranking, and the deficiency removal can be evaluated. Thus, the reference implementation of CoCoME helps us to answer Q1–Q6. Similar to our evaluation on Palladio Fileshare, the SOFA implementation of CoCoME answers Q1 and Q2. Furthermore, the results of both CoCoME implementations can be compared with each other.

The design deficiencies we searched for were Transfer Object Ignorance (see Sect. 3.2), Interface Violation, Unauthorized Call, and Inheritance Between Components. The design deficiency Interface Violation describes a situation

in which an interface is bypassed. In contrast, Unauthorized Call is an invocation of an operation which is provided by an interface but which may not be called because the calling component is not connected to the called component. Furthermore, according to Szyperski, inheritance over several components violates the component encapsulation principle, thus Inheritance Between Components is another deficiency Archimatrix is able to detect. More detailed descriptions of these deficiencies are given on the Archimatrix Web site [1].

11.1 Palladio fileshare

Palladio Fileshare realizes a server-based file sharing platform. As such, it represents a typical business information system and its component-based architecture is well documented [34]. Palladio Fileshare consists of 51 classes with almost 5,400 lines of code altogether. According to the documentation, it consists of five components.

For the validation, we first executed an architecture reconstruction with SoMoX. The used configuration is described on the Archimatrix Web site [1]. The architecture reconstruction on the whole system took 9 s.¹

Figure 14 shows the components that were recovered. The primitive components are labeled with a numerical tag starting with PC, the composite components are labeled with tags starting with CC. The numerical tags are generated and assigned by the architecture reconstruction algorithm during the creation of the components of the architecture model. Interfaces and connectors are omitted in this figure for a better readability, although they are part of the formal model.

¹ The duration of the analysis steps was measured on a machine with an Intel Core i7-2620M processor with 2.7 GHz and 6 GB RAM.

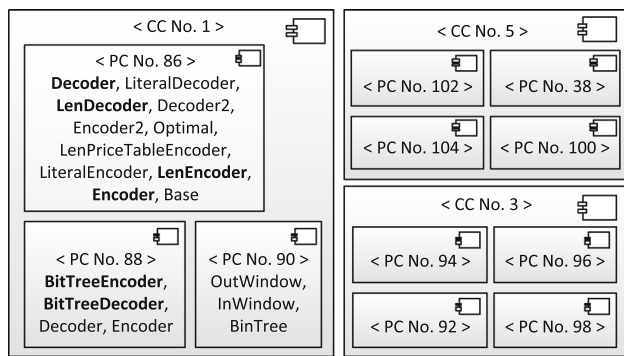


Fig. 14 Reconstructed architecture for Palladio Fileshare

For the primitive components inside CC No. 1, Fig. 14 also visualizes the contained classes because they are relevant in our next evaluation step.

As shown in Fig. 14, three composite components (labeled with CC) and eleven primitive components (labeled with PC) were detected in Palladio Fileshare.

In the next step, the component relevance analysis was performed. It only took one second and identified the component CC No. 5 as the most relevant and CC No. 1 as the second most relevant component for design deficiency detection.

Subsequently, we executed a design deficiency detection with Reclipse. The detection took 2 min and 52 s. As a result, we detected 11 occurrences of the Transfer Object Ignorance deficiency. Some of them are located in the class **LenEncoder** which was assigned to the primitive component PC No. 86. There, methods of the class **BitTreeEncoder** in the component PC No. 88 are called and objects of the type **Encoder** are passed as parameters. Classes that are involved in the detected deficiency occurrences are marked in boldface in Fig. 14. A manual inspection verified that **Encoder** is not a typical data class since it contains several methods that are neither getters nor setters. Instead, they contain more complex application logic. The same situation occurs for the call of the class **BitTreeDecoder** and a parameter of the type **Decoder**. Consequently, the communication between these components violates the component-oriented design principle that transfer objects have to be used for interactions between different components. In the deficiency removal step, we removed the detected deficiencies one by one through manual reengineering. Subsequent architecture reconstructions showed that the reconstructed architecture did not change. Therefore, we conclude that in case of this case study, the detected deficiencies did not influence the metric values enough to influence the architecture reconstruction.

11.2 Reference implementation of CoCoME

CoCoME represents a component-based trading system. Its well-documented architecture is intended to illustrate good

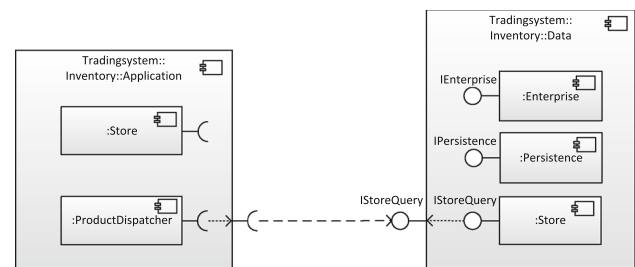


Fig. 15 An excerpt of the conceptual architecture of CoCoME based on [27]

component-oriented design. CoCoME's reference implementation was created manually by a group of computer science students. It consists of 127 classes with more than 5,000 lines of code.

Figure 15 shows an excerpt of the conceptual architecture of CoCoME as documented in [27]. According to the documentation, the component TradingSystem::Inventory::Data is designed to consist of the three sub components Enterprise, Persistence, and Store. Those subcomponents are not supposed to communicate with each other. The composite component TradingSystem::Inventory::Application can access the Store subcomponent of the composite component Data via an interface. The latter components are also used by other components which have been omitted in this excerpt.

In the initial clustering of the CoCoME system, six primitive components and six composite components were reconstructed in 13 s. A visualization of the reconstructed architecture is shown in Fig. 16. In addition to the components' names from the clustering, we manually mapped the names of the corresponding components from the documented architecture to the recovered components by comparing the contained classes. The names are displayed in parentheses. For example, the two primitive components PC No. 90 and PC No. 46 can be mapped to the Inventory::Data component from the architecture that is documented for CoCoME. Note that composite component CC No. 11 contains all other reconstructed components. It therefore corresponds to the documented TradingSystem component which encompasses the whole system.

The reconstructed architecture differs from the documented architecture because the component Data is split into two components (PC No. 46 and PC No. 90) which belong to different composite components (CC No. 1 and CC No. 3).

In the next step, we performed the component relevance analysis. Table 1 lists the detected components in combination with their relevance values. The first column represents the components' names given during the clustering. The second column shows the relevance results from the component relevance analysis for the component. The component rele-

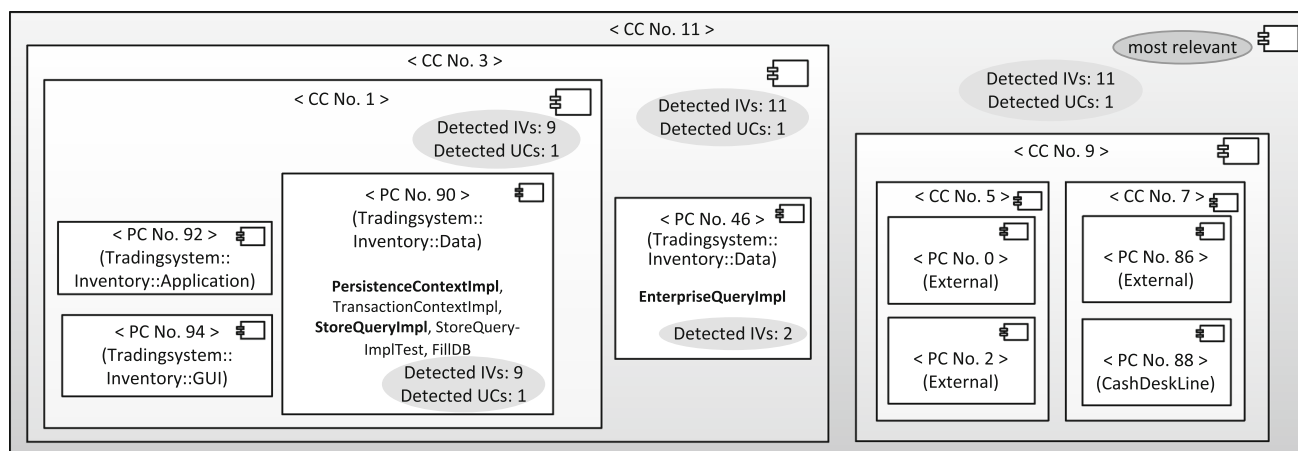


Fig. 16 Reconstructed architecture resulting from a clustering of CoCoME

Table 1 Component relevance analysis results for CoCoME

Component	Relevance
CC No. 11	0.7398
CC No. 3	0.4375
CC No. 1	0.4246
CC No. 9	0.3025
CC No. 7	0.2772
PC No. 86	0.2574
PC No. 92	0.1897
PC No. 94	0.1551
PC No. 90	0.0832
CC No. 5	0.0254
PC No. 88	0.0200
PC No. 46	0.0129

vance analysis took 13 s. The three composite components that received the highest relevance ratings are CC No. 11, CC No. 3, and CC No. 1.

In the next step, we applied the design deficiency detection. According to the results of the component relevance analysis, CC No. 11 is the most relevant component. However, as CC No. 11 encompasses the whole system, we chose the second most relevant component, CC No. 3, to be searched for design deficiencies. Additionally, we executed a design deficiency detection on the whole system and compared the results as well as the runtime.

Table 2 shows the detection results. Column 1 lists the four different design deficiencies. The second column shows the results for the analysis of component CC No. 3 and the third column shows the results for the analysis on the whole system.

The analysis of CC No. 3 took 2:26 min and the analysis of the whole system took 6:25 min.

Table 2 Detected design deficiencies in the reference implementation of CoCoME

Design deficiency	Detected occurrences	
	CC No. 3	Whole system
Transfer Object Ignorance	0	0
Interface Violation	11	11
Unauthorized Call	1	1
Inheritance Between Components	0	0
Runtime (in min)	02:26	06:25

We detected eleven Interface Violations and one Unauthorized Call which were all identified as true positives by manual inspection.

Figure 16 also shows the number of deficiencies that were detected per component. As depicted there, all deficiencies were detected in the composite component CC No. 3. They are distributed among the subcomponents CC No. 1, PC No. 90, and PC No. 46. In CC No. 1 and PC No. 90, we detected nine Interface Violations and the Unauthorized Call. In component PC No. 46, two Interface Violation occurrences were detected.

The eleven occurrences of the design deficiency Interface Violation are listed in Table 3. These occurrences all concern the interface PersistenceContext and the method getEntityManager in its subclass PersistenceContextImpl.

Two occurrences (#1 and #2) are located in the class EnterpriseQueryImpl and for the other occurrences, the accessing class is StoreQueryImpl. In the design deficiency ranking (runtime: one second), the occurrences in EnterpriseQueryImpl received a higher ranking (0.5562) than the other ones (0.4047). Their relevance values are Pareto optimal. As depicted in the last column of Table 3, the Interface Violation occurrences that received the Pareto optimal relevance values involve two components (PC No. 46 and PC No. 90)

Table 3 Detected Interface Violation occurrences in CoCoME and their ranking

	Roles				Ranking	Corresponding components
	Interface	AccessedMethod	AccessingClass	AccessingMethod		
#1	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getQueryEnterpriseById	0.5562 (opt.)	PC 46 & PC 90
#2	PersistenceContext	getEntityManager	EnterpriseQueryImpl	getMeanTimeToDelivery	0.5562 (opt.)	PC 46 & PC 90
#3	PersistenceContext	getEntityManager	StoreQueryImpl	queryAllStockItems	0.4047	PC 90
#4	PersistenceContext	getEntityManager	StoreQueryImpl	queryLowStockItems	0.4047	PC 90
#5	PersistenceContext	getEntityManager	StoreQueryImpl	queryOrderById	0.4047	PC 90
#6	PersistenceContext	getEntityManager	StoreQueryImpl	queryProductById	0.4047	PC 90
#7	PersistenceContext	getEntityManager	StoreQueryImpl	getStockItems	0.4047	PC 90
#8	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItemById	0.4047	PC 90
#9	PersistenceContext	getEntityManager	StoreQueryImpl	queryStoreById	0.4047	PC 90
#10	PersistenceContext	getEntityManager	StoreQueryImpl	queryProducts	0.4047	PC 90
#11	PersistenceContext	getEntityManager	StoreQueryImpl	queryStockItem	0.4047	PC 90

while the other occurrences are located entirely within the component PC No. 90.

The detected Unauthorized Call occurrence concerns the same classes and is also located in the method `getEntityManager`.

The Interface Violation occurrence #1, being one of the two most relevant Interface Violations, was selected to be removed automatically by extending the interface PersistenceContext.

Figure 17 depicts the previewed architecture for the application of the selected removal strategy. The deficiency removal from the GAST including the creation of the architecture preview took 6s. In comparison with the originally reconstructed architecture, in the previewed architecture, the two data components PC No. 46 and PC No. 90 belong to the same composite component (CC No. 1). The same result occurred for the removal of Interface Violation occurrence #2 which received the same ranking as occurrence #1.

After performing several iterations of the reengineering process, in which all other Interface Violation occurrences

were removed successively, the previewed architecture did not change again.

In a next evaluation step, the results were previewed for beginning the reengineering with the removal of the Interface Violation occurrence #3, one of the less relevant design deficiency occurrences, instead of selecting one of the two more relevant deficiencies. The removal was also accomplished by extending the interface. This time, the previewed architecture did not change. The same applies for the removal of Interface Violation occurrences #4 to #11.

11.3 SOFA implementation of CoCoME

As described above, we also analyzed another implementation of CoCoME and compared whether it contains significantly fewer design deficiencies than the reference implementation and how this affects the clustering. Therefore, we applied Archimatrix to the SOFA implementation of CoCoME which consists of 153 classes and approximately 7,000 lines of code.

The results of the clustering on the SOFA implementation of CoCoME are depicted in Fig. 18. Seven composite components and 23 primitive components were recovered. The clustering took 27 s.

The component relevance analysis evaluated composite component CC No. 13 to be the most relevant component for design deficiency detection. It took 13 s.

Accordingly, we first executed a design deficiency detection on CC No. 13 and then, for comparison, on the whole system containing all components. Table 4 shows the results.

The analysis on the whole system took nearly an hour, while the analysis on CC No. 13 took only a few minutes. We detected two similar Interface Violation occurrences in

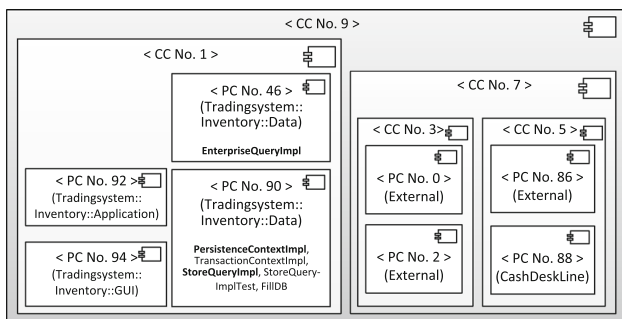


Fig. 17 Architecture preview for a design deficiency removal in CoCoME

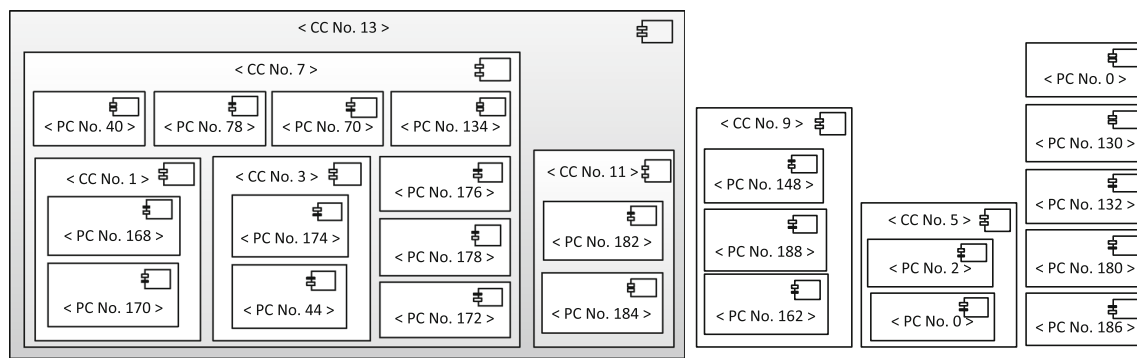


Fig. 18 Reconstructed architecture for the SOFA implementation of CoCoME

Table 4 Detected design deficiencies in the SOFA implementation of CoCoME

Design deficiency	Detected occurrences	
	CC No. 13	Whole system
Transfer Object Ignorance	0	0
Interface Violation	2	2
Unauthorized Call	1	1
Inheritance Between Components	1	1
Runtime (in min)	03:27	54:33

the class `AbstractReportingServiceClient` in the component `PC No. 174` (which is contained in `CC No. 13`). The manual inspection showed that the class `ReportProxy` that contains the accessed method is a data class, which makes these violations variants of the Interface Violation deficiency.

We also detected one occurrence of the Unauthorized Call design deficiency. The class `DataDownloadAction` calls one method of the class `DataExchangeClientObjectUpdater` in another component. This is not allowed since the first component does not require an interface provided by the latter component.

Furthermore, we detected an occurrence of the design deficiency Inheritance Between Components.

11.4 Time and effort

In summary, the execution of the different analysis steps only took a few minutes for each system. The reported durations are just the analysis durations without the time needed for the loading of the analyzed models, e.g., the GAST. Archimatrix uses the EMF framework which is not designed for model loading performance. Thus, approximately 1–2 h have to be added for the loading of the models, depending on the size of the system.

For the architecture reconstruction step, the software architect has to determine a suitable clustering configuration. This is a non-trivial task which requires some experi-

ence. However, since an architecture reconstruction run only takes a few seconds for each of the analyzed systems and since there is an approved default configuration to start with, finding a suitable configuration can be accomplished within about 1 h.

Furthermore, the time for the documentation and formalization of deficiencies (Step 3 in Fig. 3) has to be taken into account. As described in Sect. 4.2, this is a complex task, which can take several hours including the validation of the formalized deficiencies by performing detections on test and real systems. However, this is an effort that must only be done once. Once a deficiency has been formalized, it can be reused over and over again. Often, only small adaptations are necessary afterward.

All in all, we needed approximately 1 day per system for the application of Archimatrix for our validation. Of course, the exact effort depends on a large number of different factors, including the system under analysis, the training of the software architect using Archimatrix, and also the quality of the tool integration into an IDE and the appropriateness of the user interface. In order to measure and analyze these factors, a controlled experiment would be necessary. In such an experiment, two groups of software architects would have to perform the same reengineering task, one group with the help of Archimatrix and one without it. Measuring the time needed for the reengineering, the quality of the architect's work, and eliciting the architect's personal thoughts with the help of questionnaires, could yield more detailed and exact insights for this topic. However, given that the manual investigation of large systems is practically impossible, even without having performed such an experiment, we conclude that Archimatrix can offer substantial support for the software architect in an acceptable time frame.

11.5 Discussion

In the following, the validation results are discussed with respect to the research questions defined at the beginning of this section.

Do the design deficiencies defined by us occur in practice in component-based system implementations even if the systems under analysis were developed in a strictly component-based way? (Q1) Our validation shows that the design deficiencies presented in this paper do occur in practice, which answers research question Q1. The reference implementation of CoCoME was created manually and although the design documentation is very detailed and clear, numerous deficiencies exist in the implementation. The same holds for Palladio Fileshare. Even the implementation of CoCoME that uses the framework SOFA contains several design deficiencies. It stands to reason that these problems will be even more significant in larger, more complex systems.

Are our deficiency formalizations sufficiently precise to detect actual deficiencies instead of false positives, i.e., can we detect deficiencies with a high precision? And are the formalizations sufficiently general to detect all occurrences of the respective deficiencies, i.e., do we achieve a high recall? (Q2) The selected systems were expected to represent a good component-based design. Accordingly, we assumed not to find many deficiency occurrences in those systems. A large number of detection results could have been indicative of imprecise formalizations. However, we did detect only a handful of deficiency occurrences in the analyzed systems (11 deficiencies in Palladio Fileshare, 12 in the reference implementation of CoCoME, 4 in the SOFA implementation). Manual inspection ensured that the detected occurrences are real design problems and not false positives. Therefore, our validation results confirm our expectations and request question Q2 can be answered positively, too.

The deficiency formalizations were created iteratively according to the process depicted in Fig. 4. By manually analyzing the systems, we made sure that all deficiencies that we could find were also detected automatically. Hence, our approach does not produce false negatives and therefore achieves a recall of 100% in our case studies.

Is the calculated component relevance value a good indicator of components in which the detection of design deficiencies is worthwhile? (Q3) The research question Q3 cannot be answered with a clear “yes.” The composite components in the reference implementation of CoCoME with a high relevance value (CC No. 3) contained deficiencies (eleven Interface Violations and one Unauthorized Call). But there were also components with a low relevance value that contained deficiencies (PC No. 90 and PC No. 46). In the analysis on Palladio Fileshare, the components that contain the detected design deficiencies are contained in the component CC No. 1, which received the second highest relevance ranking. However, being a heuristic, a high relevance of a component does not guarantee the occurrence of design deficiencies. Although complexity is, for example, proven to be an

indicator of the occurrence probability of defects, there may still be cases in which complex components do not contain deficiencies while simple ones are full of them (for example, because they may have been developed by an inexperienced programmer). Considering that the relevance analysis deliberately uses these heuristics to get a quick estimation of the component’s probability to contain deficiencies, we think that, the result of the relevance analysis is still satisfying. The exact, and therefore much slower, detection of deficiencies is tackled in the next step of the Archimatrix process, after all.

How does the limitation of the scope for the deficiency detection by the relevance analysis improve scalability with respect to pure pattern matching? (Q4) Comparing the run times of the analyses on the whole systems with the analyses only on the most relevant component plus the run times of the component relevance analyses, it is notable that reducing the search scope to the most relevant component results in a significant saving of time for the process in all regarded systems. The observed performance gain in the analysis of CoCoME showed a factor of 3 for the reference implementation and a factor of 16 for the SOFA implementation. As the run time of the deficiency detection increases exponentially with the size of the system under analysis, it stands to reason that this speedup will be even more substantial for larger systems. This allows a positive answer to research question Q4.

In the component relevance analysis performed on CoCoME, the largest component is rated as most relevant because both available metrics depend on the size of the component. Since the components clustered with SoMoX are often all contained in one composite component [41], this should be taken into account in the component relevance analysis in future work, for example by ignoring the outermost component.

Does the removal of the deficiencies that receive a high ranking value lead to architectural changes and does the removal of deficiencies with a low ranking value leave the architecture unchanged, i.e., do the deficiency ranking heuristics work? (Q5) The validation on the reference implementation of CoCoME has shown that the removal of design deficiencies that were ranked highly led to architectural changes. In contrast, the removal of deficiencies with lower ranking values did not result in a changed architecture. Consequently, research question Q5 can be answered positively.

Is the reconstructed architecture after the removal of a relevant design deficiency closer to the documented architecture? (Q6) The reconstructed architecture that was the result of the removal of a highly ranked design deficiency still differed from the originally recovered architecture in the assignment

of the Data component. In the reengineered version, however, the components corresponding to the documented Data component were in the same composite component while this was not the case in the original architecture. Consequently, the modified architecture is closer to the documented architecture than the architecture before the removal. This means that the design deficiency ranking correctly identified design deficiency occurrences whose removal lead to an architecture that is closer to the documented architecture, than the originally reconstructed architecture. Design deficiency occurrences whose removal did not change the architecture received a lower ranking. These findings allow a positive answer to research question Q6.

11.6 Threats to validity

The validity of the presented case studies might be impacted by several factors. One threat to internal validity might be the selection of existing tools for the architecture reconstruction (SoMoX) and deficiency detection (Reclipse). Different tools might have yielded different results, leading to different conclusions. However, this threat is in part mitigated by the fact that both tools have been used in a number of other works and can be considered as reliable. In addition, the concepts implemented by the chosen tools are also represented in other available software. For example, practically all clustering-based software architecture reconstruction tools use a combination of different metrics to achieve their results.

One threat to the external validity is of course the selection of case studies in the validation. Compared to industrial business information systems, the selected systems are rather small. However, due to several reasons why we chose them.

First, in order to judge whether the architectural changes caused by the deficiency removal really lead to architectures that are closer to the conceptual architecture, a thorough architectural documentation is necessary (see Sect. 2). This is hard to come by in general but was available for our case studies.

In addition, the case studies are viable, albeit small, representatives of realistic business information systems: Palladio Fileshare exhibits a typical client–server architecture while CoCoME was intentionally designed as a benchmark business information system.

Concerning the scalability, the small size of the case studies allows only for limited conclusions. Still, the results indicate that Archimatrix will be applicable to larger systems: The architecture reconstruction has been shown to scale to systems with over 250,000 LOC [29]. Since the component relevance analysis calculates its results based on the available metric values from the architecture reconstruction, it will scale linearly with the size of the system under analysis. Finally, the other steps, deficiency detection, ranking, and removal, are not directly dependent on the size of the system

as they are only performed on a number of selected components. Their dependence on the size of that selection is addressed in Sect. 11.7.

Therefore, we believe that the case studies are able to deliver meaningful insights.

11.7 Limitations

In summary, we are able to answer most of our research questions positively. However, our approach is still subject to several limitations.

Existence of a component-based architecture First of all, we assume that the architecture of the system under analysis is a least component-based to a certain degree. In a purely monolithic system, the clustering is not capable of recovering a reasonable initial architecture [33]. If only one or several very large components were reconstructed, Archimatrix would still be able to detect deficiencies. However, the performance gain by focusing the pattern detection to a selection of components would be marginal to nonexistent. Therefore, the software architect would have to configure SoMoX such that the reconstructed components are not too large (or he would have to select appropriate subcomponents, cf. the analysis of the reference implementation of CoCoME). If that is not possible, Archimatrix will not be able to perform a faster deficiency detection than other detection approaches.

Interdependence of deficiency occurrences A problem with the removal of design deficiencies is that if one occurrence has been removed, others may be invalidated. For example, in the reference implementation of CoCoME, all detected Interface Violation occurrences are concerned with the interface PersistenceContext. As soon as one of the occurrences is removed by extending that interface, all other occurrences can be removed more easily (by using the new interface). The different deficiency occurrences are related to one another. Thus, it seems sensible to add support to remove such groups of related design deficiency occurrences altogether at the same time. This could also be considered in the deficiency ranking. If an interface is bypassed several times (maybe even in the same class), the removal of these deficiencies may be more critical than the removal of an Interface Violation where the interface is only bypassed once.

Use of heuristics It is also important to note that the clustering and with it also the component relevance analysis and the design deficiency ranking are based on heuristics. They are intended to support the software architect, but the final decisions are intentionally left to the architect. The architect also has to take part in the configuration of the clustering because the clustering depends on weights for the used metrics. Another set of metric weights often leads to a different

recovered architecture [8,33]. This limitation has been inherited from SoMoX which we did not change but just reuse.

12 State of the art and related work

Because Archimatrix uses several techniques in its reengineering process, there is a broad range of related work. Fig. 19 gives an overview of those related research areas and shows how Archimatrix relates to them.

The three main research areas that Archimatrix is concerned with are software architecture reconstruction, pattern detection (or, more specifically, deficiency detection), and refactoring and reengineering. Each of these areas plays an important role in our approach but publications which are concerned with just one of these areas are only remotely related to Archimatrix. Therefore, we do not give an overview of publications which lie in one of these areas.

Combinations of any two of these areas are more closely related. Approaches which combine software architecture reconstruction and pattern detection are presented in Sect. 12.1. Many publications deal with the detection of design problems and also suggest techniques to remove the detected problems. These are summarized in Sect. 12.2. Approaches which are concerned with the reconstruction of architectures and their subsequent reengineering are treated in Sect. 12.3. In addition, we describe works that are related to the relevance analysis and the deficiency ranking of Archimatrix in Sect. 12.4.

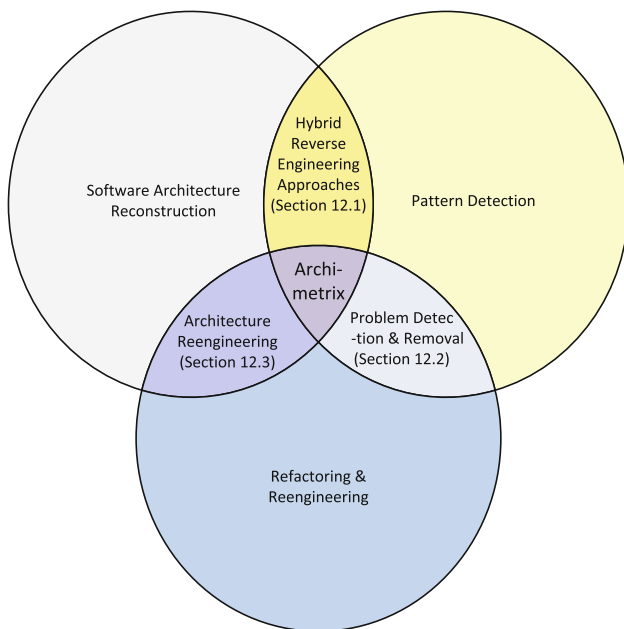


Fig. 19 Overview of related research topics

12.1 Hybrid reverse engineering approaches

In this section, we present approaches that combine different reverse engineering techniques like metrics, clustering, or pattern detection. In addition, we consider work which employs pattern-based techniques for the reconstruction of software architectures.

Keller et al. [28] 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 [8] in our approach are in line with the more rigorous component definition by Szyperski [51].

Tzerpos and Holt note that structural properties of the system under analysis should be considered during clustering [57]. Therefore, they define a number of “subsystem patterns” which are detected by their clustering algorithm. These patterns are hard-coded and are not meant to be extended by the user of their clustering tool. They do not consider deficiency patterns.

Mancoridis et al. [35] present a web-based portal site which provides a number of reverse engineering tools to its visitors. Users can upload their own source code and select different analysis methods such as clustering, code browsing, code metrics, or visualization. They deliberately do not suggest a specific process for the combination of the tools as they want to leave this decision to the user.

Sartipi [47] 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.

Bauer and Trifu [7] 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 [6] 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 relationship between the two parts nor do they suggest multiple iterations of their approach. The detection of pre-defined patterns is not addressed.

Munro [39] as well as Salehie et al. [45] 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.

Binkley et al. [10] present the concept of *Dependence Anti Patterns*, dependence structures in source code which may have negative effects on program comprehension, reverse engineering, and maintenance. The authors define a set of seven dependence anti-patterns and use a combination of program slicing and metric analysis to detect them. They do not consider the removal of these anti-patterns. Neither do they investigate the precise influence of the anti-patterns on the aforementioned software engineering tasks.

Similar to our approach, Arcelli Fontana and Zanoni [4] use an AST representation of a system as a common basis for pattern detection and architecture reconstruction. However, they use the two techniques independently of each other but do not combine them.

Arendt et al. [5] present a quality assurance process which uses a combination of metrics and structural patterns to identify model smells. They combine the smells with pre-defined graph transformations to provide an automated refactoring for identified model 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.

12.2 Problem detection and removal

Similar to clustering-based reverse engineering, there are a lot of approaches for the detection of software patterns. An exhaustive overview of different approaches in pattern-based reverse engineering in general is given by Dong et al. [18]. Design deficiencies at the architecture level as presented in this paper are strongly related to the code level “bad smells” introduced by Fowler [23]. In this section, we focus on approaches that try to detect and remove these bad smells in source code.

Tahvildari and Kontogiannis use metrics to measure, for example, the coupling and cohesion in a system in order to find deficiencies [52]. Then, they remove these deficiencies with appropriate meta-pattern transformations and re-evaluate the metrics. They argue that applying the transformations improves the metric values and therefore the quality of the system. They do not present a concept for the rating of deficiencies nor do they analyze where to begin with the deficiency detection.

Tourwé and Mens detect “refactoring opportunities,” identify matching transformations and execute them automatically [53]. The bad smells and refactorings considered in their work are at a very low level of abstraction (e.g., identifying and removing obsolete parameters). The impact of the refactorings on the system architecture is not in their focus.

In [56], 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 design deficiencies are detected to speed up the detection and also reduce the number of results. In addition, we provide an automatic deficiency ranking that removes the need to validate the deficiencies manually.

Stürmer et al. [50] model guideline violations in Matlab models and their automatic correction. For the violation detection, they also use Reclipse. Thus, because they do not guide the detection process by a component relevance analysis, they suffer the drawback of high detection durations.

Moha et al. [38] present the DECOR approach for the specification and detection of bad smells. After the detection, a manual validation of detection results and manual refactoring of detected bad smells are proposed. Our approach puts the emphasis on where to look for the deficiencies and what to do with them after the detection.

The report by Zhang et al. [64] gives an overview of the current knowledge on bad smells and states that there still is little empirical evidence available that removing code level bad smells is actually worth the effort. The deficiencies targeted in our paper, however, are not the typical bad smells but are deficiencies on an architectural level.

None of the aforementioned approaches considers the impact of deficiencies on the architecture reconstruction or provides a means to guide the deficiency detection as we do in the component relevance analysis. In addition, no preview of the reengineering impact on the architecture is carried out.

12.3 Architecture reengineering

Krikhaar presents an architecture improvement process [31] in which an “ideal” architecture is constructed manually. The existing software is then analyzed with respect to import relations, part of hierarchies and use relations at code level. The “ideal” and the “reverse-architected” architecture are then to be compared manually to identify violations. Actions to remove violations are not discussed in the paper. The author also suggests to incorporate code metrics in future work. In follow-up work, Krikhaar et al. [32] present a two-phase process for the improvement of software architectures. Here, a model is generated from code. The software architect has to

manually evaluate this model and think of ideas to improve it. Then, “recipes” to apply the ideas to the code are created manually. Finally, the architect has to implement automatic transformations for the created recipes. The impact of the improvement ideas can be evaluated on the model by using metrics or “box-and-arrow” diagram visualization. In our approach, the improvement opportunities are automatically identified by detecting deficiencies. Automated transformations can be provided for the deficiencies and an automated architecture preview can analyze and visualize the impact of the transformation.

Bianchi et al. [9] present a process to iteratively reengineer a complete system without shutting it down. The process supports the iterative migration of functionality and data. The authors propose to first break down a system into components that can then be reengineered individually. In contrast to the approach in this paper, neither the identification of components nor the reengineering is the focus of their work. Both steps are assumed to be carried out manually.

Sarkar et al. [46] report on their endeavor to break up a legacy banking application into components. The presented process includes the creation of a modular architecture, semi-automatic identification of architecture violations, a manual refactoring step, and manually implemented checks (“gatekeeper tools”) to enforce compliance with the reengineered architecture. Archimetrix provides substantially more support to the software architect through automated process steps such as the deficiency ranking or the architecture preview.

Erdmenger et al. [21] describe SOAMIG, an iterative, generic process model for the migration of legacy code to a service-oriented architecture. Similar to our approach, the process contains reverse engineering steps for legacy code analysis and refactoring steps for the improvement of the migrated system. In addition, the authors try to automate as much of the migration as possible. However, as their approach is meant to be generic, they do not specifically define how, e.g., the target architecture for the migration should be obtained. They also do not focus on the detection of deficiencies in the legacy system. Our ideas could perhaps be incorporated in a specific instance of the SOAMIG process model.

Frey and Hasselbring [25] present CloudMIG, an approach which provides a process to reengineer legacy applications for the cloud. The authors put an emphasis on aspects like resource efficiency and scalability of the target architecture. Although the reconstruction of the original architecture is mentioned in their work, it is not focused on. During the migration, the CloudMIG approach can be used to identify elements of the target architecture that violate constraints of the cloud environment. In contrast, our approach concentrates on revealing deficiencies that do not stem from a migration but from the long-term evolution of the system.

12.4 Relevance analysis and deficiency ranking

Simon et al. [48] are among the first to argue that tool support is necessary to point out where refactorings can be applied. They suggest to use metrics and an appropriate visualization to help developers find worthwhile refactoring locations. However, they admit that their approach does not scale for large applications. The execution of the refactoring and the impact on the system are not in the focus of their work.

Marinescu adds a filtering mechanism to his metric-based bad smell detection approach that determines which occurrences are relevant for further processing [36]. This approach also uses the composition of several metrics to detect design deficiencies. In the filtering mechanism, detected occurrences with extreme metric values or values that are in a particular range are searched. He does not cover the removal of detected bad smells.

Bourquin and Keller present an approach that is focused on manual refactorings on the architecture level [12]. They argue that bad smells “cannot be quantified easily, and therefore are hard to prioritize.” They manually analyze the relevance of their refactorings on the architecture after their application. To analyze the refactoring results, they use code metrics and a comparison between the number of detected bad smells before and after the refactoring.

13 Conclusions and future work

In this paper, we presented Archimetrix, a tool-supported architecture reconstruction process for component-based systems. It uses clustering techniques to reconstruct the architecture of legacy systems. In addition, it takes design deficiencies in these systems into account by providing the means to detect and remove these deficiencies. In our validation, we applied Archimetrix to different real-life systems: two variants of the Common Component Modeling Example and the Palladio Fileshare system. We were able to show that these systems contain deficiencies and that detecting and removing them is facilitated by Archimetrix. We also showed that the deficiencies have an impact on the clustering-based reconstruction of the system architecture.

In the future, Archimetrix could be extended to support more scenarios from the field of software architecture evolution and maintenance.

At the moment, the software architect has to decide when to end the Archimetrix process by looking at the reconstructed architecture. To support this, it might be useful to integrate a calculation of architecture quality metrics into the process. Every time, deficiencies are removed and a new architecture is reconstructed, those metrics could be evaluated. The architect could then choose to execute the

Archimetrix process until the quality metrics have reached a certain threshold.

Another idea is to use Archimetrix for the conformance checking of architecture and implementation. Once the architect arrives at a suitable architecture by executing the Archimetrix process that architecture could serve as a reference point for future extensions. When a developer extends the software, an architect could reconstruct its architecture with Archimetrix, compare it to that reference point, and notify the developer of a possible mismatch. To this end, Archimetrix could also be incorporated into a continuous testing and integration environment. On the conceptual level, our architecture reconstruction approach SoMoX is already capable of considering existing architectures. Leveraging this capability would be a valuable addition for Archimetrix.

Acknowledgments We thank Oleg Travkin for his contributions in the development of Archimetrix. We would also like to thank Christian Heinzemann, Dietrich Travkin, and the anonymous reviewers for their valuable comments. This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (CRC 901).

References

1. Archimetrix: <http://www.fujaba.de/archimetrix.html> (2013)
2. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, 2nd edn. Addison-Wesley, Boston (2006)
3. Alur, D., Crupi, J., Malks, D.: *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall/Sun Microsystems Press, Upper Saddle River (2001)
4. Arcelli, Fontana F., Zanoni, M.: A tool for design pattern detection and software architecture reconstruction. *Inf. Sci.* **181**(7), 1306–1324 (2011)
5. Arendt, T., Kranz, S., Mantz, F., Regnat, N., Taentzer, G.: Towards syntactical model quality assurance in industrial software development: process definition and tool support. In: *Proceedings of the Software Engineering*. Springer, Berlin (2011) (to appear)
6. Basit, H.A., Jarzabek, S.: Detecting higher-level similarity patterns in programs. *SIGSOFT Softw. Eng. Notes* **30**(5), 156–165 (2005)
7. Bauer, M., Trifu, M.: Architecture-aware adaptive clustering of OO systems. In: *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, pp. 3–14. ACM (March 2004)
8. Becker, S., Hauck, M., Trifu, M., Krogmann, K., Kofron, J.: Reverse engineering component models for quality predictions. In: *Proceedings of the 14th European Conference on Software Maintenance and Reengineering on IEEE Computer Society*, pp. 199–202 (2010)
9. Bianchi, A., Caivano, D., Marengo, V., Visaggio, G.: Iterative reengineering of legacy systems. *Trans. Softw. Eng.* **29**(3), 225–241 (2003)
10. Binkley, D., Gold, N., Harman, M., Li, Z., Mahdavi, K., Wegener, J.: Dependence anti patterns. In: *Proceedings of the 4th International ERCIM Workshop on Software Evolution and Evolvability on IEEE*, pp. 25–34 (2008)
11. Boehm, B., Basili, V.R.: Software defect reduction top 10 list. *IEEE Comput.* **34**(1), 135–137 (2001)
12. Bourquin, F., Keller, R.K.: High-impact refactoring based on architecture violations. In: *Proceedings of the 11th European Conference on Software Maintenance and Reengineering on IEEE*, pp. 149–158 (2007)
13. Brown, W.J., Malveau, R.C., McCormick, H.W., Mombay, T.J.: *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York (1998)
14. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: a taxonomy. *IEEE Softw.* **7**(1), 13–17 (1990)
15. Cho, E.S., Kim, M.S., Kim, S.D.: Component metrics to measure component quality. In: *8th Asia-Pacific, Software Engineering Conference*, pp. 419–426 (December 2001)
16. Chouambe, L., Klatt, B., Krogmann, K.: Reverse engineering software-models of component-based systems. In: *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008) on IEEE Computer Society*, pp. 93–102, Athens (2008)
17. Coello Coello, C., Dhaenens, C., Jourdan, L.: Multi-objective combinatorial optimization: problematic and context. In: *Advances in Multi-objective Nature Inspired Computing*, vol. 272 of *Studies in Computational Intelligence*, pp. 1–21. Springer, Berlin (2010)
18. Dong, J., Zhao, Y., Peng, T.: A review of design pattern mining techniques. *Int. J. Softw. Eng. Knowl. Eng.* **19**(6), 823–855 (2009)
19. Ducasse, S., Pollet, D.: Software architecture reconstruction: a process-oriented taxonomy. *IEEE Trans. Softw. Eng.* **35**(4), 573–591 (2009)
20. Eppstein, D.: Subgraph isomorphism in planar graphs and related problems. In: *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms on Society for Industrial and Applied Mathematics*, pp. 632–640. Philadelphia (1995)
21. Erdmenger, U., Fuhr, A., Herget, A., Horn, T., Kaiser, U., Riediger, V., Teppe, W., Theurer, M., Uhlig, D., Winter, A., Zillmann, C., Zimmermann, Y.: The SOAMIG process model in industrial applications. In: *Proceedings of the 15th European Conference on Software Maintenance and Reengineering on IEEE*, pp. 339–342 (2011)
22. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story aiagrams: a new graph rewrite language based on the unified modeling language and java. In: *Selected Papers from the 6th International Workshop on Theory and Application of Graph Transformations*, vol. 1764 of *Lecture Notes in Computer Science*, pp. 296–309. Springer, Berlin (2000)
23. Fowler, M.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading (1999)
24. Fowler, M.: *Patterns of Enterprise Application Architecture*. Addison-Wesley, Reading (2002)
25. Frey, S., Hasselbring, W.: Model-based migration of legacy software systems to scalable and resource-efficient cloud-based applications: the CloudMIG approach. In: *Proceedings of the First International Conference on Cloud Computing, GRIDs and Virtualization*, pp. 155–158. Xpert Publishing Services (November 2010)
26. Glass, R.L.: *Facts and Fallacies of Software Engineering*. Addison-Wesley, Reading (2003)
27. Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolok, H., Mirandola, R., Hummel, B., Meisinger, M., Pfaller, C.: CoCoME—the common component modeling example. In: *The Common Component Modeling Example*, vol. 5153 of *Lecture Notes in Computer Science*, pp. 16–53. Springer, Berlin (2008)
28. Keller, R.K., Schauer, R., Robitaille, S., Pagé, P.: Pattern-Based Reverse-Engineering of Design Components. In: *Proceedings of the 21st International Conference on Software Engineering*, pp. 226–235. IEEE Computer Society Press (May 1999)
29. Koziolok, H., Schlich, B., Bilich, C., Weiss, R., Becker, S., Krogmann, K., Trifu, M., Mirandola, R., Koziolok, A.: An industrial case study on quality impact prediction for evolving service-oriented

- software. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE 2011), Software Engineering in Practice Track. ACM Press (2011)
30. Kpodjedo, S., Ricca, F., Galinier, P., Guéhéneuc, Y.-G., Antoniol, G.: Design evolution metrics for defect prediction in object oriented systems. *Empir. Softw. Eng.* **16**(1), 141–175 (2011)
 31. Krikhaar, R.: Reverse architecting approach for complex systems. In: Proceedings of the 13th International Conference on Software Maintenance on IEEE, pp. 4–11 (1997)
 32. Krikhaar, R., Postma, A., Sellink, A., Stroucken, M., Verhoef, C.: A two-phase process for software architecture improvement. In: Proceedings of the 15th International Conference on Software Maintenance on IEEE, pp. 371–380 (1999)
 33. Krogmann, K.: Reconstruction of Software Component Architectures and Behaviour Models Using Static and Dynamic Analysis. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe (2010)
 34. Krogmann, K., Kuperberg, M., Reussner, R.: Using genetic search for reverse engineering of parametric behavior models for performance prediction. *IEEE Trans. Softw. Eng.* **36**(6), 865–877 (2010)
 35. Mancoridis, S., Souder, T.S., Chen, Y.-F., Gansner, E.R., Korn, J.L.: REportal: a web-based portal site for reverse engineering. In: Proceedings of the 8th Working Conference on Reverse Engineering on IEEE, pp. 221–230 (2001)
 36. Marinescu, R.: Detection strategies: Metrics-based rules for detecting design flaws. In: Proceedings of the 20th International Conference on Software, Maintenance, pp. 350–359 (September 2004)
 37. McCabe, T.J.: A coomplexity measure. *IEEE Trans. Softw. Eng.* **SE-2**(4), 308–320 (1976)
 38. Moha, N., Guéhéneuc, Y.-G., Duchien, L., Le Meur, A.-F.: DECOR: a method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.* **36**(1), 20–36 (2010)
 39. Munro, M.J.: Product metrics for automatic identification of “Bad Smell” design problems in java source-code. In: 11th IEEE International Symposium on Software Metrics, pp. 15–23 (September 2005)
 40. Niere, J., Schäfer, W., Wadsack, J., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: Proceedings of the 24th International Conference on Software Engineering, pp. 338–348. ACM Press (2002)
 41. Platenius, M.C.: Reengineering of design deficiencies in component-based software Architectures. Master’s Thesis, University of Paderborn (October 2011)
 42. Platenius, M.C., von Detten, M., Becker, S.: Archimatrix: improved software architecture recovery in the presence of design deficiencies. In: Proceedings of the 16th European Conference on Software Maintenance and Reengineering on IEEE, pp. 255–264 (March 2012)
 43. Rausch, A., Reussner, R., Mirandola, R., Plasil, F.: The Common Component Modeling Example—Comparing Software Component Models, vol. 5153 of Lecture Notes in Computer Science. Springer, Berlin (2008)
 44. Riel, A.J.: Object-oriented design Heuristics. Addison-Wesley, Reading (1996)
 45. Salehie, M., Li, S., Tahvildari, L.: A metric-based heuristic framework to detect object-oriented design flaws. In: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC 2006), IEEE Computer Society, pp. 159–168 (2006)
 46. Sarkar, S., Ramachandran, S., Kumar, G.S., Iyengar, M.K., Rangarajan, K., Sivagnanam, S.: Modularization of a large-scale business application: a case study. *IEEE Softw.* **26**(2), 28–35 (2009)
 47. Sartipi, K.: Software architecture recovery based on pattern matching. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, pp. 293–296 (2003)
 48. Simon, F., Steinbrückner, F., Lewerentz, C.: Metrics based refactoring. In: Proceedings of the 5th Conference on Software Maintenance and Reengineering on IEEE, pp. 30–39 (2001)
 49. Sissy—Structural Investigation in Software Systems: <http://www.sqools.org/sissy/> (2013)
 50. Stürmer, I., Kreuz, I., Schäfer, W., Schür, A.: The MATE approach: enhanced simulink/stateflow model transformation. In: Proceedings of the MathWorks Automotive Conference (June 2007)
 51. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
 52. Tahvildari, L., Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations. In: Proceedings of the 7th European Conference on Software Maintenance and Reengineering, pp. 183–192 (March 2003)
 53. Tourwé, T., Mens, T.: Identifying refactoring opportunities using logic meta programming. In: Proceedings of the 7th European Conference on Software Maintenance and Reengineering, pp. 91–100 (March 2003)
 54. Travkin, O.: Kombination von Clustering- und musterbasierten Reverse-Engineering-Verfahren. Master’s thesis, University of Paderborn (June 2011) (in German)
 55. Travkin, O., von Detten, M., Becker, S.: 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 (February 2011)
 56. Trifu, A., Seng, O., Genssler, T.: Automated design flaw correction in object-oriented systems. In: Proceedings of the 8th Conference on Software Maintenance and Reengineering, IEEE, pp. 174–183 (2004)
 57. Tzerpos, V., Holt, R.C.: ACDC: an algorithm for comprehension-driven clustering. In: Proceedings of the 7th Working Conference on Reverse Engineering, WCRE on IEEE, pp. 258–267 (2000)
 58. von Detten, M.: Reengineering of Component-Based Software Systems in the Presence of Design Deficiencies. PhD thesis, Heinz Nixdorf Institute, University of Paderborn, Germany (2013)
 59. von Detten, M., Becker, S.: Combining clustering and pattern detection for the reengineering of component-based software systems. In: Proceedings of the 7th International Conference on the Quality of Software Architectures, pp. 23–32. ACM (June 2011)
 60. von Detten, M., Heinzemann, C., Platenius, M.C., Rieke, J., Travkin, D., Hildebrandt, S.: Story diagrams—syntax and semantics. Technical report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, Ver. 0.2, University of Paderborn (July 2012)
 61. von Detten, M., Meyer, M., Travkin, D.: Reclipse—a reverse engineering tool suite. Technical report tr-ri-10-312, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn (March 2010)
 62. von Detten, M., Meyer, M., Travkin, D.: Reverse engineering with the reclipse tool suite. In: Proceedings of the 32nd International Conference on Software Engineering, pp. 299–300. ACM (May 2010)
 63. von Detten, M., Travkin, D.: 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 (October 2010)
 64. Zhang, M., Hall, T., Baddoo, N.: Code bad smells: a review of current knowledge. *J. Softw. Maint. Evol. Res. Pract.* **23**(3), 179–202 (2011)

Author Biographies



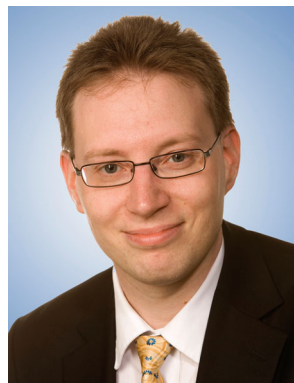
Markus von Detten is a research assistant at the Software Engineering Group at the Heinz Nixdorf Institute at the University of Paderborn. He received his PhD in software engineering with highest honors from the University of Paderborn in 2013. His thesis is titled “Reengineering of Component-Based Software Systems in the Presence of Design Deficiencies” and the origin of the Archimatrix project. His research interests include reverse engineering and reengi-

neering of business information systems, software quality, and service matching. He is also a member of the Collaborative Research Centre 901 “On-The-Fly Computing.”



Marie Christin Platenius has been a PhD student at the Software Engineering Group at the Heinz Nixdorf Institute at the University of Paderborn since October 2011. She received her master’s degree in computer science at the University of Paderborn. Her master’s thesis was titled “Reengineering of Design Deficiencies in Component-Based Software Architectures.” Currently, she is a research assistant in the Collaborative Research Center 901

“On-The-Fly Computing” funded by the German Research Foundation (DFG). Her research interests include service-oriented software engineering, component-based software engineering, and reengineering.



Steffen Becker is a Junior-professor for Software Engineering with a focus on model-driven software engineering at the University of Paderborn since April 2010. His current research projects tackle self-adaptive software systems and scalable cloud applications. Before, he was a department manager in the software engineering department of the FZI Research Centre for Information Technology, Karlsruhe, where he supervised several industry projects related to

architectural quality and acted as project manager of the EU FP7 project Q-ImPRESS with a focus on the quality of evolving service-oriented software systems. He received his PhD in software engineering from the University of Oldenburg in 2008 for his work on the quality prediction of model-driven developed software systems in the context of the five year DFG-funded young researchers group “Palladio.” He is also a member of various program committees of conferences in that area. Especially, he has been a permanent member of the steering committee of the International Conference on the Quality of Software Architectures (QoSA) since 2005 which is now part of the CompArch federated event.