Towards the Combination of Clustering-based and Pattern-based Reverse Engineering Approaches

Oleg Travkin, Markus von Detten, Steffen Becker
Software Engineering Group, Heinz Nixdorf Institute
Department of Computer Science, University of Paderborn, Paderborn, Germany
[oleg82|mvdetten|steffen.becker]@mail.uni-paderborn.de

Abstract: Reverse Engineering, i.e. the analysis of software for the purpose of recovering its design documentation, e.g. in form of the conceptual architecture, is an important area of software engineering. Today, two prevalent reverse engineering approaches have emerged: (1) the clustering-based approach which tries to analyze a given software system by grouping its elements based on metric values to provide the reverse engineer with an overview of the system and (2) the pattern-based approach which tries to detect predefined structural patterns in the software which can give insight about the original developers' intentions. These approaches operate on different levels of abstraction and have specific strengths and weaknesses. In this paper, we sketch an approach towards combining these techniques which can remove some of the specific shortcomings.

1 Introduction

During their life cycle, software systems have to be maintained and continuously extended. In order to perform these tasks, a developer has to first understand the design of the software which can be a tedious and complicated task due to incomplete design documentation. Reverse engineering is the task of analyzing a software system in order to recover its design documentation. In the past, two types of approaches for the reverse engineering of software systems have been proposed: clustering-based reverse engineering and pattern-based reverse engineering [Sar03].

Clustering-based reverse engineering approaches try to identify the system's architecture by grouping its basic elements (e.g. classes) based on the values of certain metrics. Examples of common metrics used for this purpose are coupling and stability [Mar94]. The result of a clustering-based analysis is a set of packages or components (plus optionally their dependencies or connectors) which represents a possible system architecture. Because the calculation of metric values is relatively easy for most metrics, a clustering-based analysis can be carried out in short time frames, even for large systems. A major drawback of this approach, however, is the high level of abstraction that comes with the use of metrics. Some complex architecture-relevant information cannot easily be captured by metric formulas. In addition, considering more details also complicates the calculation which would greatly increase the time consumption of the approach. An overview of clustering-based reverse engineering approaches is given for example by Ducasse and Pollet [DP09].

Pattern-based reverse engineering approaches are concerned with the detection of predefined patterns in the software. In this paper, we only consider structural patterns. Patterns exist for different levels of abstraction, different domains and purposes but the most famous collection of patterns are the *Design Patterns* by Gamma et al. [GHJV95]. The rationale behind their detection is that each design pattern has a specific intent that describes in which situation the pattern should be applied. Hence, the detection of a design pattern can reveal the developer's design intentions and thereby foster the understanding of the software. Another renowned category of patterns are the Anti Patterns [BMMM98] which describe frequently occurring bad solutions to common problems together with refactorings to correct them. The detection of patterns in general is the subject of many scientific approaches of which Dong et al. give an overview [DZP09]. Because pattern detection often takes much more detailed information into account than clustering-based approaches, it is generally much slower. Moreover, due to their low level of abstraction, an impractically large set of pattern implementations can be detected even for medium-scale systems.

In order to combine the strengths and overcome the limitations of both approaches, we propose a reverse engineering process which first performs a metric-based clustering for a given system to identify components and afterwards applies a pattern detection mechanism to each of the identified components.This combination mainly addresses two problems. It improves the pattern-based analysis by providing a more focused approach than a complete analysis of the whole system and thereby reduces the required time and the number of detection results. Second, the higher level of detail that is taken into account during a pattern-based analysis can be utilized to refine the architecture recovered by the clustering-based analysis: The pattern detection can identify certain patterns which have an influence on the metrics used by the clustering and which can therefore affect the recovered architecture. An example of this is presented in Section 3.

We intend to build a prototype reverse engineering tool by combining the Software Model Extractor SoMoX developed at the FZI Karlsruhe [BHT+10] and the pattern detection tool suite Reclipse developed at the University of Paderborn [vDMT10]. While SoMoX uses a weighted combination of metrics to infer the architecture of industry size systems, Reclipse is capable of detecting arbitrary structural patterns whether they are design or anti patterns.

The contribution of this paper is a reverse engineering approach that combines clustering-based and pattern-based analyses to improve the reverse engineered results and ultimately provide more accurate architectural information. The following section briefly presents the proposed combined detection process. Section 3 shows an example which benefits from this combination. In Section 4 we draw conclusions and sketch future work.

2 The combined reverse engineering process

Figure 1 presents the main steps of the combined reverse engineering process and the different artifacts involved in it. The two inputs of the clustering step are the source code of the system and a set of metrics. The result is the heuristically recovered architecture

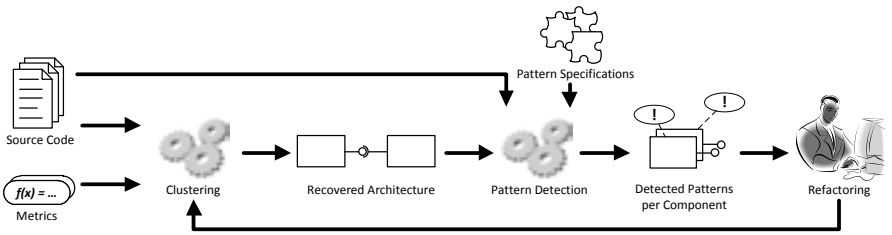

Figure 1: Illustration of the combined reverse engineering process

of the software. The source code together with the recovered architecture and a set of structural pattern specifications then serves as an input for the pattern detection algorithm. Because the pattern detection is scoped on the detected components, the required analysis time and the number of results are obviously reduced. On the other hand, patterns that exist across component boundaries are neglected by the scoped approach. This problem is subject of future work. The detected patterns can serve as a starting point for refactorings which can have an impact on the architecture of the software. Hence, a clustering of the refactored system may lead to a different and possibly improved recovered architecture.

3 Possible influences of anti patterns on the conceptual architecture

One of our basic assumptions is that that the existence of certain patterns can have an influence on the architecture recovered by the clustering. For the sake of brevity, we restrict our discussion in this paper to anti patterns.

As the clustering relies on code metrics like coupling and stability, anti patterns which affect these metrics can adulterate the clustering results. One example for this is an illegal direct access, an anti pattern which increases the coupling between two classes. This is illustrated in the following example.

The upper left part of Figure 2 shows exemplary source code for a simple system with two interfaces IA and IB and two classes A and B which implement those interfaces. IA declares a method m1 and IB declares a method m2. Class A implements IA and has a reference to IB. Class B, implementing IB, additionally declares a method m3. The implementation of m1 in class A downcasts the object contained in the field ib to the type B and then calls m3, thereby circumventing the use of the interface. Part b) of the figure shows a class diagram representation of the same system which clarifies that A references both, IB and B. The coupling between A and be is therefore 1.0 [1].

The use of the concrete type instead of the interface violates one of the programming principles defined by Gamma et al.: *"Program to an interface, not an implementation."*

[1] The coupling between A and B is computed as the ratio between the accesses from A to B and the number of accesses from A to all classes if the system. In this simple, illustrative example $Coupling(A, B) = \frac{1}{1} = 1$.

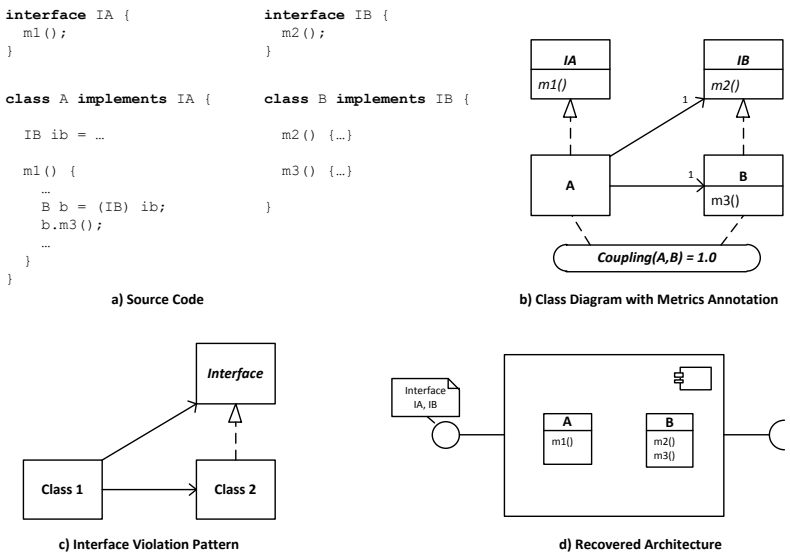

Figure 2: Violation of an interface and the resulting component

[GHJV95]. It also leads to a tight coupling between classes A and B which should have been prevented by the use of interfaces. This is a design flaw known as *Interface Violation* which is depicted in the lower left part of Figure 2. The existence of this flaw can have several reasons: for example the declaration of m3 could simply be missing in IB or m3 may not supposed to be accessed from the outside. Either way, the high value of the coupling metric would probably cause clustering-based architecture recovery tools to group A and B together, for example as members of the same software component as shown in part d) of Figure 2.

Figure 3 shows a refactored version of the code from Figure 2. Method m3 has been added to the interface IB so that the downcast in class A becomes unnecessary. The two classes now communicate directly via their implemented interfaces which reduces their direct coupling to 0. The class diagram representation of the refactored code clarifies that A no longer has a direct reference to B.

The refactoring removes the interface violation from the system. Because this design change affects the coupling of A and B, it can have an impact on the clustering of the system. An architecture recovery tool might now group the classes as depicted in the lower part of Figure 3. A and B would be assigned to two different software components which communicate solely via their interfaces, i.e. the component containing class B provides the interface IB which is required by the component containing A.

The detection and removal of the interface violation consequently leads to a change in the reverse engineered architecture. We were able to detect several of the described interface violations in the reference implementation of the Common Component Modeling Example

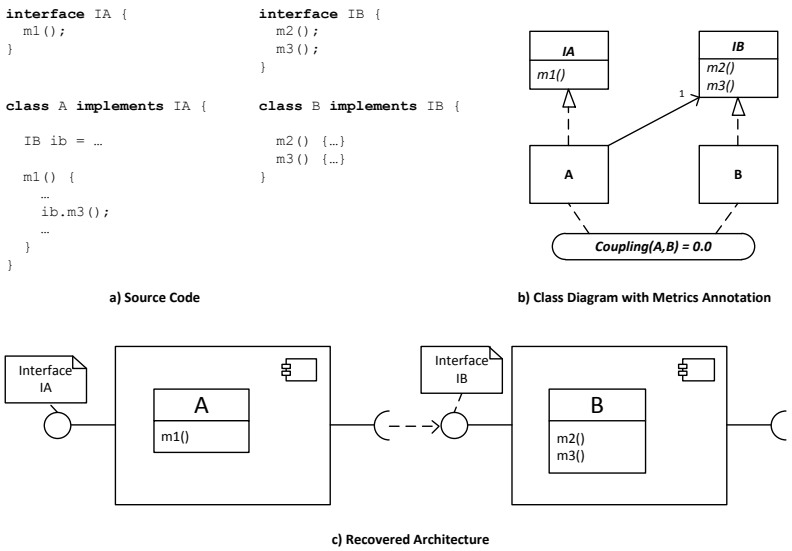

Figure 3: Refactored code with the improved component structure

CoCoME [HKW+08] which could prove to have significant impact on the clustering of the implementation with current reverse engineering tools (e.g. SoMoX [BHT+10]).

4 Conclusions and Future Work

We sketched an approach which combines clustering-based and pattern-based reverse engineering techniques and suggested that this combination can have multiple advantages. Pattern-based reverse engineering approaches can be used for the identification of (anti) patterns that affect the clustering. The removal of these patterns can then improve the recovered architecture. Second, we argued that a clustering of the system prior to the detection of patterns could enhance the focus and speed of the detection and would not overwhelm the reverse engineer with too many results.

Our research is still in its early stages and numerous points remain to be considered. First, we want to identify more patterns which can have an influence on the architecture recovery. It must also be noted that not all occurrences of a pattern necessarily have the same impact. Interface violations between two classes are only critical if the two classes conceptually belong to different components. Classes in the same package may as well be expected to be tightly coupled. We suppose that these cases can be distinguished by a relevance analysis step that takes additional metrics into account. In the example, an Interface Violation could only be marked as critical if the two classes reside in different packages and are otherwise coupled to independent sets of other classes.

As mentioned in Section 2, the confinement of the pattern detection to the detected components results in false negatives which has to be considered in future work.

We also plan to extend the reverse engineering process by additional automated steps that help the reverse engineer with further design decisions. For example, when a critical Interface Violation has been detected, it could be removed in several different ways by automatic transformations (which extend the interface, remove the violating access, etc.). These transformations can have different impacts on the clustering which could be analyzed and presented to the reverse engineer to improve his knowledge about the choices at hand.

In addition, it would be interesting to evaluate different implementations of the CoCoME example. We discovered several interface violations in the reference implementation but implementations with component frameworks like SOFA or FRACTAL may have different issues. This could help to understand to which extent the choice of an implementation framework affects the quality of the final implementation.

References

[BHT+10] S. Becker, M. Hauck, M. Trifu, K. Krogmann, and J. Kofron. Reverse Engineering Component Models for Quality Predictions. In *Proc. of the 14th European Conference on Software Maintenance and Reengineering*, pages 199–202. IEEE, 2010.

[BMMM98] W. J. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mombray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1998.

[DP09] S. Ducasse and D. Pollet. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[DZP09] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *Int. Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, 2009.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HKW+08] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, M. Meisinger, and C. Pfaller. CoCoME - The Common Component Modeling Example. In *The Common Component Modeling Example*, volume 5153 of *LNCS*, pages 16–53. Springer, 2008.

[Mar94] R.C. Martin. OO Design Quality Metrics - An Analysis of Dependencies. 1994.

[Sar03] K. Sartipi. Software Architecture Recovery based on Pattern Matching. In *Proc. of the Int. Conference on Software Maintenance*, pages 293–296. IEEE, 2003.

[vDMT10] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proc. of the 32nd Int. Conference on Software Engineering*, volume 2, pages 299–300. ACM, 2010.