Reverse Engineering with the Reclipse Tool Suite

Markus von Detten, Matthias Meyer, Dietrich Travkin
Software Engineering Group, Heinz Nixdorf Institute
Department of Computer Science, University of Paderborn, Germany
[mvdetten|mm|travkin]@upb.de

ABSTRACT

Design pattern detection is a reverse engineering methodology that helps software engineers to analyze and understand legacy software by recovering its design and thereby aiding in the preparation of re-engineering activities. We present RECLIPSE, a reverse engineering tool suite for static and dynamic design pattern detection in combination with a pattern candidate rating used to assess the detection results' reliability.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, Reverse Engineering, and Reengineering*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms

Languages, Algorithms

1. INTRODUCTION

Due to requests for new features and the discovery of defects, software has to be continuously extended and adapted during its life cycle. Incomplete documentation often complicates maintenance.

The tedious and error-prone task of understanding a large software system can be supported by tools that recover the system's design by locating instances of design patterns [1]. Identifying pattern instances can reveal the original developers' design intentions. This paves the way for re-engineering activities by identifying adaptable or exchangeable software parts, and presents adequate modification alternatives.

Several approaches for the detection of design pattern implementations have been developed, most of which apply a static, structural source code analysis. Some patterns, however, also possess characteristic behavior. Furthermore, patterns can be structurally similar to others (e.g. *Strategy* and *State* [1]). A purely static analysis fails to detect or distinguish those patterns or yields too many false positives.

Consequently, we developed a pattern detection approach that integrates a static, structural analysis with a subsequent, dynamic runtime behavior analysis. The static analysis is used to detect potential pattern instances [3], i.e. *candidates*. Where necessary, the dynamic analysis is used to verify the runtime behavior of the candidates [5, 6].

In order to support the detection of new patterns and to enable the analysis of arbitrary software systems or models, we offer a flexible, graphical specification language based on an exchangeable meta-model for structural and behavioral patterns. This way, we also applied the approach to the detection of design deficiencies in Java code [2] and guideline violations in MATLAB Simulink models.

We cover numerous pattern implementation variants in our structural pattern specifications by complementing a pattern's invariant core structure with non-mandatory, additional constraints which, if satisfied, increase the candidate's probability to be a true positive. An automated rating mechanism quantifies a candidate's number of satisfied, additional constraints and thereby its compliance to its pattern specification on a percentage basis. Thus, the most promising candidates can be discerned by their rating.

In this paper we present the approach and especially its implementation as a plug-in for the Eclipse IDE: the RECLIPSE Tool Suite[1] [4]. We focus on the specification of implementation variants, the candidate rating and the dynamic analysis.

2. THE PATTERN DETECTION PROCESS

Our pattern detection process is illustrated in Figure 1. A reverse engineer specifies structural patterns with special object diagrams. In a static analysis step, the system's source code is analyzed and parts which comply to the structural patterns are detected. These structures are likely to constitute pattern instances, thus, they are called *candidates*. Afterwards, in case of patterns with a characteristic behavior, a dynamic analysis is applied to confirm or reject the candidates depending on their observed runtime behavior.

Static Analysis

For the static analysis, an abstract syntax graph representation of the software under analysis is created from the source code. Formally, a structural pattern specification defines a graph transformation rule which tries to match a certain structure and adds an annotation to it. An inference algo-

[1] http://www.fujaba.de/reclipse

ICSE '10, May 2-8 2010, Cape Town, South Africa

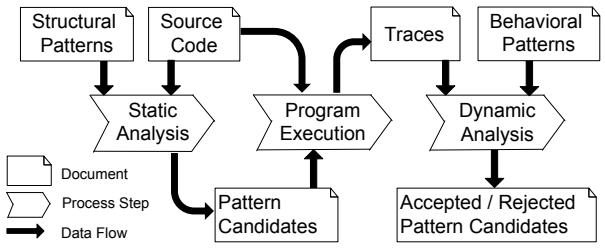

Figure 1: The pattern detection process

rithm then applies the different rules to the abstract syntax graph and tags parts whose structure complies to a pattern with annotations. For details we refer to [3].

Pattern Variants and Rating of Pattern Candidates

In order to enable the recognition of different pattern variants with a single specification, we introduce the concept of *additional constraints*. They allow to specify conditions whose satisfaction is desired but not mandatory to constitute an actual pattern instance.

For example, many patterns describe certain properties, like the visibility of an attribute or a class being abstract, which are not essential and are therefore often neglected in implementations. Hence, we mark such constraints as *additional*. This way, we detect both, candidates with and without the specified properties.

We use the information given by the satisfaction of additional constraints to separate reliable from less reliable results. This is done automatically by rating each pattern candidate with a percental value that relates the number of constraints satisfied by a candidate to the total number of constraints in the corresponding pattern specification. The more constraints are satisfied the higher is the rating value.

The rating quantifies the degree of a pattern candidate's compliance to its specification and helps the reverse engineer to distinguish true from false positives. Moreover, a threshold can be set in order to display only pattern candidates with a rating higher than the threshold.

Dynamic Analysis

The dynamic analysis is based on traces of the pattern candidates. To obtain these traces, the system under analysis is executed and method calls between instances of classes which are part of a candidate are recorded. Note that we only trace the candidates' behavior instead of the whole system which drastically reduces the search space for the dynamic analysis.

The expected behavior for a given pattern is formally specified with special sequence diagrams. They describe interactions between elements in structural patterns and determine which method calls may occur in which order between the classes that participate in a pattern candidate.

During the analysis step, the traces are compared to the corresponding behavioral patterns. For this purpose, a special automaton is automatically generated for each behavioral pattern. If a trace matches the pattern, it is accepted and rejected otherwise [7].

If the majority of a candidate's traces match the behavioral pattern, the candidate likely is an actual design pattern instance. If most of the traces for a candidate do not match the behavioral pattern, it is assumed to be a false positive which is only structurally similar to the actual pattern.

3. EVALUATION RESULTS

We used our prototype implementation in the RECLIPSE Tool Suite to evaluate the approach by analyzing, e.g. Java's Abstract Window Toolkit, the Java Generic Library, Eclipse's Standard Widget Toolkit, and JUnit 3.8.2. Details on our results are given in [4].

We compared the analysis results of JUnit to other approaches. In accordance with them we found instances of the Composite, Decorator and Template Method patterns. We also detected several Observer candidates, but the dynamic analysis revealed that only one of them showed a correct Observer behavior. The other candidates were false positives.

4. CONCLUSIONS AND FUTURE WORK

In this paper we presented our reverse engineering approach. The major advantages compared to other approaches are the combination of static and dynamic analysis for the detection of patterns as well as its flexible, yet formal way of structural and behavioral pattern specification that enables the re-engineer to extend and adapt the specifications.

Furthermore, the approach allows to specify several structural variants of a pattern at one go by means of additional constraints which are used to automatically rate detected pattern candidates.

We obtained evaluation results which showed that our approach and its implementation are mature and efficient enough to be applied to real systems.

In the future, we plan to allow the usage of "fuzzy" values in metric-based constraints and thereby avoid metric thresholds that are hard to determine. We also want to integrate the concept of additional constraints into the behavioral patterns and intend to provide better support for the interpretation of dynamic analysis results.

5. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[2] M. Meyer. Pattern-based Reengineering of Software Systems. In *Proc. of the* 13^{th} *Working Conference on Reverse Engineering*, pages 305–306, 2006.

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

[4] M. von Detten, M. Meyer, and D. Travkin. Reclipse – A Reverse Engineering Tool Suite. Technical Report tr-ri-10-312, University of Paderborn, Germany, 2010.

[5] M. von Detten and M. C. Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proc. of the* 7^{th} *International Fujaba Days*, pages 15–19, 2009.

[6] L. Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung (Structural and Behavioral Design Pattern Detection)*. PhD thesis, University of Paderborn, 2007. In German.

[7] L. Wendehals and A. Orso. Recognizing Behavioral Patterns at Runtime using Finite Automata. In *Proc. of the* 4^{th} *ICSE Workshop on Dynamic Analysis*, pages 33–40, 2006.