Improving Design Pattern Instance Recognition by Dynamic Analysis*

Lothar Wendehals
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
33098 Paderborn, Germany
lowende@upb.de

Abstract

Design pattern instance recognition is often done by static analysis, thus approaches are limited to the recognition of static parts of design patterns. The dynamic behavior of patterns is disregarded and leads to lots of false positives during recognition. This paper presents an approach to combine the advantages of static and dynamic analyses to overcome this problem and improve the design pattern instance recognition.

1. Motivation

Reverse engineering large industrial legacy systems is hard work. They consist of several thousand or up to million lines of code and often lack of documentation. The systems have grown over several years and were developed by different programmers with different programming styles.

Design recovery, which means extracting design documents from source code, is a way to assist the reengineer understanding and maintaining those systems. As a basis for design documentation design patterns first presented by Gamma et al. [4] are suitable. By recognizing instances of design patterns in the system's source code, the implicit design may be recovered and documented. Further enhancements can then be applied to the system.

Most approaches to design recovery use static analysis techniques on the system's source code [1, 6, 7, 12]. Some of them are text-search tools based on regular expressions. Other approaches use graph representations of the source code, such as control flow or data flow graphs or even abstract syntax trees.

In object-oriented languages those static analyses are not sufficient. Polymorphism and dynamic method binding prevent the correct analysis of method invocations that are essential to recover patterns with behavioral aspects such as the *Chain of Responsibility* pattern [4] depicted in Figure 1.

Some parts of a *Chain of Responsibility* pattern such as the inheritance between the abstract class Handler and their concrete children classes or the self-association successor of the class Handler can be found by static analyzing techniques. Method calls such as the delegation between a Handler object and its successor can be found statically, but the concrete invoked method and the concrete object the method is invoked on can only be analyzed during runtime.

Thus a precise recognition of design pattern instances with dynamic behavior requires dynamic analysis. A complete reengineering process based on dynamic analysis only is not appropriate, because static parts of design patterns can be identified easier in static analysis. So a smart combination of static and dynamic analysis is desirable.

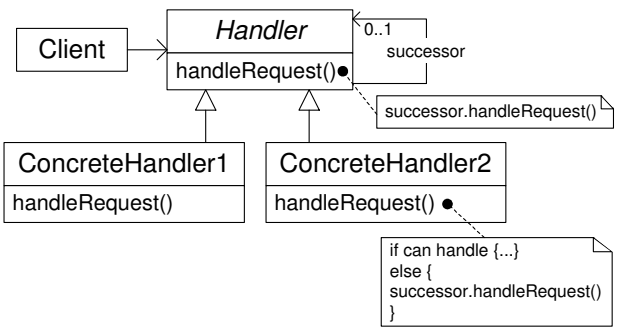

Figure 1: *Chain of Responsibility* pattern

The combined reengineering process starts with the static analysis of the source code. As a result of this first part of the process a set of pattern instance candidates is produced. This set is the input for the dynamic analysis part of the process. It reduces the search space for the dynamic analysis. During runtime of the program pattern instance candidates only have to be investigated.

In the following an overview of our pattern-based design recovery process is presented. An example for a pattern instance is then given to clarify the limitations of static analysis. To lift this restrictions dynamic analysis is added to design pattern instance recognition based on static analysis. The paper closes with related work and some conclusions.

2. Pattern-based Design Recovery

In our approach described in [8, 9] we use an abstract syntax graph (ASG) representation of the source code. This ASG is produced by parsing the source code. It contains static information about classes, attributes, methods including method bodies and inheritance. Our approach is not bound to any particular programming language. As a case study we analyzed software systems written in Java.

We use additional nodes to enrich the ASG with information gathered during analyses. Those nodes added to the ASG are called annotations. They are linked to the nodes in the ASG that have to be annotated with information.

*This work is part of the FINITE project funded by the German Research Foundation (DFG), project-no. SCHA 745/2-1.

A tool-based design recovery needs formalized rules for the analysis. We developed a graphical rule definition language based on graph-rewrite-rules with a left and a right side. Each pattern that should be searched for is defined by the left side of such a graph-rewrite-rule. The right side of the rule consists of the pattern together with the annotation node that has to be added to the ASG. By successfully applying these rules to the ASG, pattern instances are recovered. The information of a found pattern instance is stored by the annotation node linked to the ASG elements that are participating in the pattern's instance.

By defining new rules, existing rules can be reused. Simple rules may be combined to new more complex and more abstract rules. As a result a pattern rule catalog is formed where rules depend on each other.

To support reverse engineering tasks where up to million lines of code are analyzed we developed a highly scalable design recovery process. We showed that our approach is applicable to real life software systems such as the Java Abstract Windowing Toolkit (AWT) [11] with more than 140.000 lines of code [8, 9].

3. Static Analysis

Pattern-based design recovery is a deductive analysis problem where patterns, or rules, are repeatedly applied to a representation of the source code to arrive at the most complete characterization of the code permitted by the rules. Pure deductive analysis algorithms typically apply the rules involved level by level - bottom-up - according to their natural hierarchy. Results from other researchers, such as [13] and [10], suggest that a reverse engineering tool providing fully automatic analysis based on this approach cannot scale for larger software systems.

We developed a combined bottom-up and top-down strategy. The rules in the pattern rule catalog are sorted by their natural dependency hierarchy. The analysis starts in bottom-up mode with rules at the lowest level which are rules that do not depend on others. After successfully applying such a rule, consequent rules at the next level will be triggered. If any rule depends on precondition rules that have not yet applied, the strategy switches into the top-down mode. After evaluating all preconditions the strategy changes back to bottom-up mode. The whole analysis algorithm which ensures a highly scalable process can be found in [8].

In our ASG representation of the source code method bodies are also contained as mentioned before. This enables our static analysis to analyze parts of the dynamic behavior of methods. The existence of method calls can be identified but dynamic method binding and polymorphism prevents to identify the actual called method and the actual object the method is invoked on. It can only be a first indication of dynamic behavior.

Figure 2 depicts an instance of a *Chain of Responsibility* pattern shown in Figure 1. This example shows a part of a model for a graphical user interface. There is an abstract class GUIElement that implements a multiple self-reference children. Concrete subclasses of this abstract class are a Window, a Panel and a Button. They override a method from their superclass. The dotted line of the inheritance relation denotes an indirect inheritance. So there are other classes in between the inheritance hierarchy.

Suppose a pattern rule is defined to identify a *Chain of Responsibility* pattern instance as shown in Figure 1. The source code to be reengineered contains a *Chain of Responsibility* pattern instance as depicted in Figure 2. During static analysis there are some elements of uncertainty that prevent an exact matching of this pattern instance.

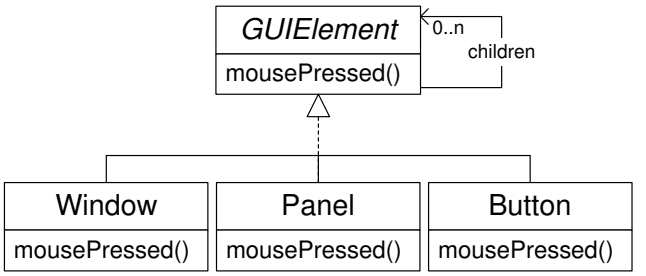

Figure 2: Concrete instance of a *Chain of Responsibility* pattern

The multiple self-reference of the abstract superclass GUIElement is different from the single reference successor of the *Chain of Responsibility* pattern. This could be a counter indicator for a *Chain of Responsibility* pattern instance, because a chain element has always only one successor. Another uncertainty derives from the indirect inheritance hierarchy. The original pattern describes a direct inheritance between the abstract handler and its concrete handlers. Furthermore the method call delegation from a handler to its successor can not be identified exactly. A method call from a handler to another handler can be statically identified, but it is not for sure that this call is forwarded in a chain of objects.

This leads only to an inexact match. There are two ways to handle this match. Firstly, this match can be discarded, because it is different from the original defined pattern. Secondly, it could be accepted as a pattern instance candidate with a low certainty of being a correct pattern instance. This certainty is expressed as a fuzzy value. In [9] we describe how to handle inexact pattern matches by fuzzy values.

The result of the overall static analysis is a set of pattern instance candidates each rated by a fuzzy value. For some of the candidates the certainty (fuzzy value) that they are actual pattern instances is not very high because of dynamic behavior that can not be analyzed statically as stated before. Some of them may even be false positives. Dynamic analysis can help to make these results more precise.

The analysis restricted to the candidates reduces the input for dynamic analysis. To further reduce the search space, our static analysis process provides the analysis of method bodies as part of the ASG. Structural information about method bodies such as a method call within a loop can be used for refining the rules. This reduces not only the number of candidates but also the number of methods that have to be investigated by dynamic analysis. Methods that are probably not participating in the pattern can be separated from those that are relevant to the pattern.

4. Dynamic Analysis

The design patterns descriptions used by Gamma et al. [4] are informal in most parts, for example the motivation, applicability, consequences and implementation. More formal parts of a pattern description are the structure and sometimes the collaboration parts. The collaboration parts often

contain UML sequence diagrams with typical behavior of the pattern constituents. Figure 3 shows such a sequence diagram for the *Chain of Responsibility* pattern. Those descriptions of the dynamic behavior of patterns can be used by the reverse engineer to formally define rules for tool-based design recovery.

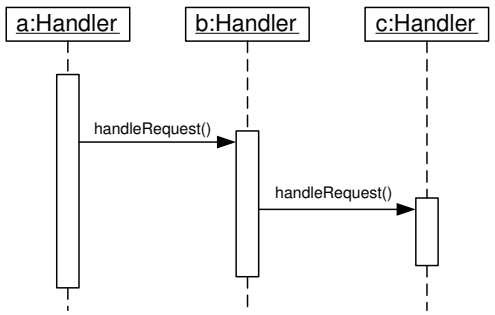

Figure 3: Sequence diagram pattern for a *Chain of Responsibility*

For each pattern with dynamic behavior a pattern for a UML sequence diagram is added to the pattern's rule. It describes typical sequences of method calls between objects that participate in the pattern. The diagrams can only be samples for object interaction. Figure 3 for example only shows three objects acting as a *Chain of Responsibility*. Actual chain of responsibilities may consist of more than three objects.

Reengineering a program often aims at changing or adding features. The program's part to be reengineered is therefore precisely defined. So the execution of the program for dynamic analysis can be restricted to those parts. The execution has to be done manually by the reengineer.

Information will be gathered during program execution by debugging the program. Basic functionality of debuggers allow to set breakpoints and record method traces. For each pattern-relevant method from candidate classes breakpoints are set. The pattern-relevant methods can be found by static analysis as mentioned before. So object information and their method traces are recorded during runtime. These information are stored as an attributed call graph and form the data for the pattern instance recognition.

The procedure of the dynamic analysis is analog to static analysis. After generating a call graph by executing the program - which corresponds to parsing the source code into an ASG in static analysis - the gathered information has to be analyzed. The sequence diagrams are defined as graph-rewrite-rules just like the static part of a pattern rule. The matching of the sequence diagrams can now be done by applying their graph-rewrite-rules to the attributed call graph, which again corresponds to applying the static pattern rules to the ASG. Finally the results of both analyses - static and dynamic - are rated by fuzzy values.

During runtime of the program there could be multiple different object sets that are instances of one pattern instance candidate. For example a *Chain of Responsibility*-pattern used in a program can be instantiated multiple times during runtime. For each of these sets object type information and method traces will be recorded. Polymorphism and dynamic method binding enables method traces of the sets to differ significantly from each other, even if they are instances of the same pattern instance candidate. In our example there could be object sets instantiated from the same *Chain of Responsibility* instance where the objects are different concrete handlers. Method traces from those sets would be different.

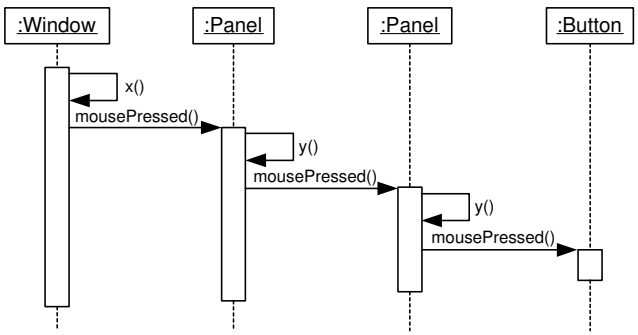

Figure 4: Method trace from a candidate object set

Figure 4 depicts an example for a object set and its method trace. These objects are instances from the class diagram of Figure 2. This object set is therefore an instance of a pattern instance candidate. There is a mouse pressed event that is delegated from a Window object to the responsible Button object. Some method calls as x() or y() may have been recorded, too. Others may have been suppressed, as method calls to different objects that were not investigated.

The matching between the pattern sequence diagram and the method trace can only be inexact. There are three objects in the pattern sequence diagram depicted in Figure 3 delegating the handleRequest() method call to their successor. This situation can be found in the method trace of Figure 4 if naming is not considered. There is one additional object and there are additional method calls that do not match any method call in the pattern. So a matching can be found but it is ambiguous and inexact. The grade of ambiguity and inexactness has to be rated for each object set and its method trace. The rating is expressed by a fuzzy value within a range between 0 and 1 like in static analysis.

Both results from static and dynamic analysis are then presented to the reengineer and has to be interpreted. There are three cases that have to be considered for each pattern instance candidate.

Firstly, there were no object set that was instantiated from the candidate during program execution. So there is only a result from static analysis. That means the features executed do not use the program's part the pattern instance candidate belongs to. Therefore the reengineer is not interested in that program's part and design and the results from static analysis can be ignored.

Secondly, there are one or more object sets with their method traces for one pattern instance candidate. In this case the fuzzy values from all object sets are combined to three values: the minimum, the average and the maximum fuzzy value.

Suppose in our example there are five object sets instantiated from the pattern instance candidate of Figure 2. Four of these sets have a fuzzy value of 0.9 and one set has a fuzzy value of 0.4. The average fuzzy value is 0.8. The static analysis result for the given example is a certainty of 0.6 of being an actual *Chain of Responsibility* pattern instance. The maximum fuzzy value from dynamic analysis confirms

this assumption. The minimum fuzzy value is a contraindication, but the average value shows that most of the fuzzy values confirm the assumption. In the case of a low average fuzzy value the result would indicate a false positive.

Thirdly, there are object sets for a pattern instance candidate, but the given sequence diagram could not be matched to the call graph. All three fuzzy values - minimum, average and maximum - will be null. This indicates that the pattern instance candidate from static analysis can be clearly identified as a false positive.

5. Related Work

Heuzeroth et al. [5] combine as well static as dynamic analysis to detect interaction patterns. They approach is similar to that presented in this paper. The source code is represented by an abstract syntax tree (AST). Static patterns are described as relations over AST node objects. The computed relations are input to the dynamic analysis. Dynamic patterns are described by protocols over a set of events. The relations as well as the protocols have to be implemented manually, which means implementing the algorithms to calculate the relations and to calculate the match of protocols. This restricts the usability of the approach because of the complicated maintenance, adaption and creation of patterns. Furthermore the approach for static analysis is limited in recognizing implementation variants of patterns. This leads either to lots of false positives or to missing pattern instances. Lots of false positives in static analysis cause then a higher complexity in dynamic analysis.

Eisenbarth et al. [2] combine static and dynamic analysis as well. Their approach helps the reengineer identifying components used for certain features. In contrast to the presented approach Eisenbarth et al. use dynamic analysis to reduce the search space for static analysis. Scenarios for all features that have to be located in the code are chosen for the program's execution. Concept analysis is then performed to identify relationships between scenarios and subprograms. These results are used in static analysis which is done by slicing techniques and manual inspection. So the search space should be small for static analysis.

6. Conclusions

An approach is presented to use dynamic program analysis to confirm results from static analysis. The static analysis as described in this paper is already implemented in our CASE tool FUJABA [3]. The implementation of the dynamic analysis is current work. Pattern rule specification and matching for dynamic analysis will be realized by graph-rewrite-rules as in static analysis. The inference algorithm [8] can therefore be reused.

We introduced the notion of fuzziness into our static analysis to rate pattern instance candidates [9]. This approach is used for the rating in dynamic analysis, too. The combination of both results from static and dynamic analysis is presented to the reengineer for each pattern instance candidate. The dynamic analysis results confirm or discard the static analysis result. Thus, the combination is a good criterion for the reliability of the results.

7. References

[1] G. Antoniol, R. Fiutem, and L. Christoforetti. Design pattern recovery in object-oriented software. In *Proc. of the* 6^{th} *International Workshop on Program Comprehension (IWPC), Ischia, Italy*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM 2001)*. IEEE Computer Society Press, November 2001.

[3] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

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

[5] D. Heuzeroth, T. Holl, and W. Löwe. Combining static and dynamic analyses to detect interaction patterns. In *Proc. of the* 6^{th} *International Conference on Integrated Design and Process Technology*, June 2002.

[6] R. Keller, R. Schauer, S. Robitaille, and P. Page. Pattern-based reverse-engineering of design components. In *Proc. of the* 21^{st} *International Conference on Software Engineering, Los Angeles, USA*, pages 226–235. IEEE Computer Society Press, May 1999.

[7] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proc. of the* 3^{rd} *Working Conference on Reverse Engineering (WCRE), Monterey, CA*, pages 208–215. IEEE Computer Society Press, November 1996.

[8] J. Niere, W. Schäfer, J. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proc. of the* 24^{th} *International Conference on Software Engineering (ICSE), Orlando, Florida, USA*, pages 338–348, May 2002.

[9] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the* 11^{th} *International Workshop on Program Comprehension (IWPC), Portland, USA*, May 2003.

[10] A. Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):84–93, May 1994.

[11] SUN Microsystems. *AWT, the SUN Java Abstract Window Toolkit. Online at http://java.sun.com/products/jdk/awt.*

[12] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proc. of the* 9^{th} *International Conference on Software Maintenance (ICSM), Oxford, UK.*, pages 230–238. IEEE Computer Society Press, September 1999.

[13] L. Wills. Using attributed flow graph parsing to recognize programs. In *Proc. of International Workshop on Graph Grammars and Their Application to Computer Science*, LNCS 1073, Williamsburg, Virginia, 1994, November 1996. Springer Verlag.