Specifying Patterns for Dynamic Pattern Instance Recognition with UML 2.0 Sequence Diagrams*

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

1 Motivation

Design recovery, which means extracting design documents from source code, assists a reverse engineer in understanding a software system. For design documentation design patterns [1] are suitable. By recognizing instances of design patterns in the system's source code the implicit design can be documented.

Most approaches use static analysis techniques e.g. [3]. In object-oriented languages static analysis only is not sufficient, since structurally similar patterns are not distinguishable from each other. Design patterns often differ only in their behavior. Polymorphism and dynamic method binding prevent a correct static analysis of method invocations that are essential to analyze behavior. Thus, for a precise recognition of design pattern instances a combination of static and dynamic analyses is reasonable, e.g. [2].

For tool supported recognition patterns have to be formally specified. Prolog is a common specification language, e.g. [2]. Others use script languages like Perl with regular expressions [3]. We developed a more intuitive graphical specification language for static analysis based on graph grammars [4].

We extend our static analysis process by dynamic analysis of method traces [6]. The process starts with static analysis of the source code resulting in a set of pattern instance candidates. The set is then verified by dynamic analysis. In the following the main focus is on a pattern specification language for behavioral patterns based on sequence diagrams.

2 Example

There are lots of design patterns that are structurally equal or at least similar such as *Decorator* and *Chain of Responsibility*, *Strategy* and *Bridge* or *Strategy* and *State*, which are depicted in Figures 1 and 2.

A *Strategy* design pattern lets an algorithm vary independently from the client that uses it. An abstract class Strategy defines the algorithm interface, which is implemented by different ConcreteStrategy classes.

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

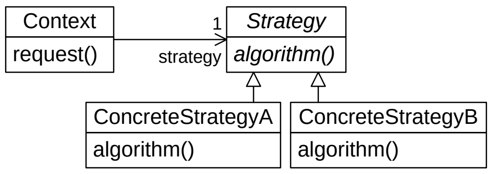

Figure 1: The *Strategy* Design Pattern

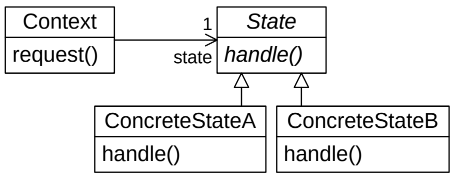

Figure 2: The *State* Design Pattern

The *State* design pattern allows an object to alter its behavior depending on its internal state. An interface for handling the behavior is defined by the abstract class State and implemented by different ConcreteStates. Implementations of these two design patterns are not distinguishable by static analysis. The next section shows how their behavior can be specified by sequence diagrams for a precise recognition.

3 Pattern Specification

In [1] there are some hints how the two design patterns *Strategy* and *State* differ in their behavior. It is said for a *Strategy* that *"A context forwards requests from its clients to its strategy. Clients usually create and pass a ConcreteStrategy object to the context..."*. So changing the concrete strategy is done by the client.

Figure 3 depicts a UML 2.0 sequence diagram for a *Strategy* that formalizes the description above. It is called a sequence pattern and can be read as follows: first of all a strategy object must be set on the context object by the client. An arbitrary number of calls from the client must be delegated to the strategy. Then the strategy may be changed by the client and another arbitrary number of delegations may be processed.

In UML 2.0 a new syntax element *Combined Fragment* is introduced to sequence diagrams. It is visualized as a rectangle with an operator within the upper

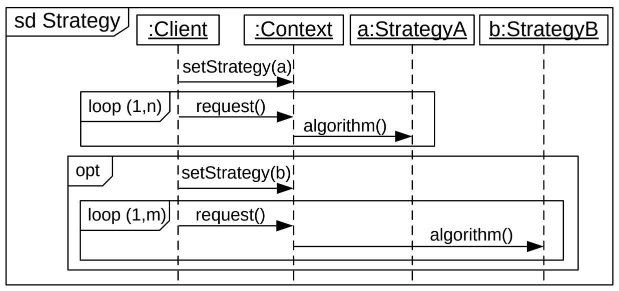

Figure 3: Sequence Pattern for *Strategy*

left corner. The operator *loop (1,n)* e.g. defines a sequence repeated at least once or up to n times. The operator *opt* defines an optional sequence. Other operators are *alternatives*, *negative*, *consider*, etc.

The operator *consider* has some methods as parameter. Only calls of those methods are displayed within the sequence, but they may be interleaved by other calls that are ignored. In sequence patterns the *consider* operator is implicitly used. Only calls of methods used within the sequence are considered in the pattern matching, other interleaving calls are ignored. In the given example there could be numerous other calls between the first setStrategy call and the loop, but no second setStrategy call. Thus, a sequence pattern does not describe a single execution sequence but a set of sequences with a common subsequence. This subsequence can be compared to some kind of slice of the program's method call trace where only a few method calls are considered.

For the *State* pattern it is said in [1]: *"... Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly. Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances."*. So the states are changed by the context or the states.

Figure 4 depicts the sequence pattern for a *State*. The state must be changed during runtime, otherwise different states make no sense. This specified as an alternative, where the state can be switched either by the current state or by the context.

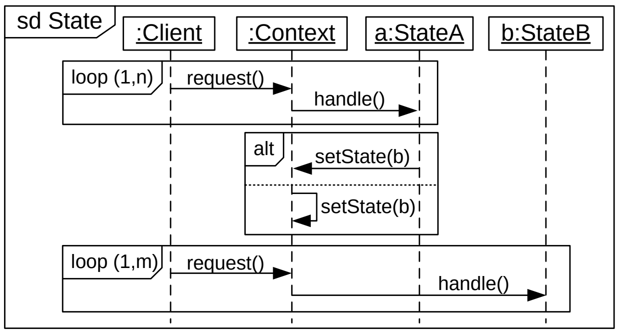

Figure 4: Sequence Pattern for *State*

The dynamic analysis uses these sequence patterns to verify pattern instance candidates from static analysis. A candidate both for *Strategy* and *State* can now be classified as one of the two patterns or as a false positive. Note, the class and method names are not considered in the analysis. They are just for a better readability of the sequence patterns.

4 Current and Future Work

In the static analysis we allow for the composition of structural patterns to new patterns to reduce the effort of specification. It has to be researched if reusing patterns is reasonable for sequence patterns, too.

The composition of patterns is especially used for higher level design patterns composed of lower level clichés. We are currently working on the specification of sequence patterns for those clichés and of course for all design patterns.

In [5] we describe how pattern instances can be rated by fuzzy values to express the accuracy of the match and to help the reverse engineer assessing the results. For sequence patterns the number of ignored calls within the matching method trace can be used to rate the match. If only a few ignored method calls interleave the given pattern sequence, the match has a high accuracy, otherwise, it has a low accuracy.

References

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

[2] D. Heuzeroth, S. Mandel, and W. Löwe. Generating design pattern detectors from pattern specifications. In *Proc. of the* 18th *International Workshop on Automated Software Engineering (ASE), Montreal, Canada*, October 2003.

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

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

[5] J. Niere, J. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *Proc. of the* 11th *International Workshop on Program Comprehension (IWPC), Portland, USA*, pages 274–279. IEEE Computer Society Press, May 2003.

[6] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *Proc. of the ICSE 2003 Workshop on Dynamic Analysis (WODA), Portland, USA*, May 2003.