# Visualization of Pattern Detection Results in Reclipse

Marie Christin Platenius, Markus von Detten, Dietrich Travkin
Software Engineering Group, Heinz Nixdorf Institute,
University of Paderborn, Paderborn, Germany
[mcp|mvdetten|travkin]@mail.uni-paderborn.de

## ABSTRACT

Reverse engineering tools can simplify the recovery of a software system's design by detecting design pattern implementations. This helps to understand a software system and thereby supports the process of maintaining or extending a software. Because the manual specification of patterns has to maintain the balance between precision and generality, detection results may contain incorrectly detected pattern implementations. Usually, a detected candidate cannot be displayed in detail so that interpreting the detection results is difficult. In this paper, we present an approach for a comprehensive and comprehensible visualization of detection results in the reverse engineering tool suite Reclipse.

## 1. INTRODUCTION

Reverse engineering is the task of analyzing a software system in order to understand its design. A helpful part in the recovery of the design is the detection of design patterns. Design patterns represent general, reusable, and commonly accepted solutions to frequently occurring problems in object-oriented software design [4]. Knowledge about the presence of pattern implementations in a software helps to understand the software system by revealing the original developers' design intentions. It thereby supports the process of maintaining or extending a software. Reverse engineering tools can automatically detect pattern implementations and thereby simplify the reverse engineering process. In the last years, the detection of design pattern implementations in source code has been the subject of many scientific publications (Dong et al. give an overview [3]).

To automate the detection, a formal specification of the patterns is needed. However, patterns can be implemented in several ways and there can be many different variants of a pattern which are difficult to capture in a single formal specification. This leads to the fact that the detection results, the so-called *pattern candidates*, can contain false positives. This problem can be mitigated to a certain degree, by specifying a mandatory pattern core which has to be present and several additional conditions whose detection increases the confidence in the correctness of a detected pattern implementation. As a consequence, the detection results must be inspected and for each candidate it has to be decided if it is a true or false positive.

The static pattern detection of the reverse engineering tool suite Reclipse[1] [7, 8] detects pattern implementations in source code. The pattern detection results are currently displayed as a simple list of pattern candidates in which the involved classes are listed. An automatically calculated percental rating value indicates how much a pattern candidate conforms to its specification. A low rating value means that the candidate does only contain the mandatory core and few or none of the specified additional conditions. Therefore, this could indicate that the candidate is a false positive.

As patterns are specified by the user, the specifications can contain (possibly subtle) flaws. Such an incorrect specification can result in erroneous detection results. Examples are false positives, or misleading rating values that are too high or too low. Currently, the user cannot distinguish these cases by looking at the list of rated detection results and she cannot see how the rating is calculated. Thus, more information about the detected candidates is needed to make use of the results.

In this paper, we present how the detection results can be visualized in a comprehensive and comprehensible way. The goal of the visualization is to provide a more detailed image of the detected pattern candidates to the user and to make the candidates' rating more transparent.

The remainder of this paper is organized as follows: First, we give a general overview of the pattern detection process, the pattern specification and the current presentation of results in Reclipse. In Section 3, requirements for a visualization of detection results are proposed and in Section 4 our visualization approach is presented. Section 5 deals with related work. We finish with conclusions and ideas for future work.

## 2. PATTERN DETECTION IN RECLIPSE

The static pattern detection in Reclipse uses a graph matching approach: The system, i.e. the source code under analysis, is represented by an abstract syntax graph (ASG) in which an inference algorithm detects subgraphs that comply to the structure of pre-specified graph patterns [5]. This results in a list of so-called *pattern candidates*. The pattern specifications consist of a number of conditions which have to be satisfied for a successful match. A percental rating value is computed for each candidate. The rating value determines the ratio of a candidate's satisfied conditions to all conditions in the corresponding pattern specification. Thereby, the rating quantifies the completeness of a candidate and, thus, indicates if the candidate is a real pattern implementation or a false positive.

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

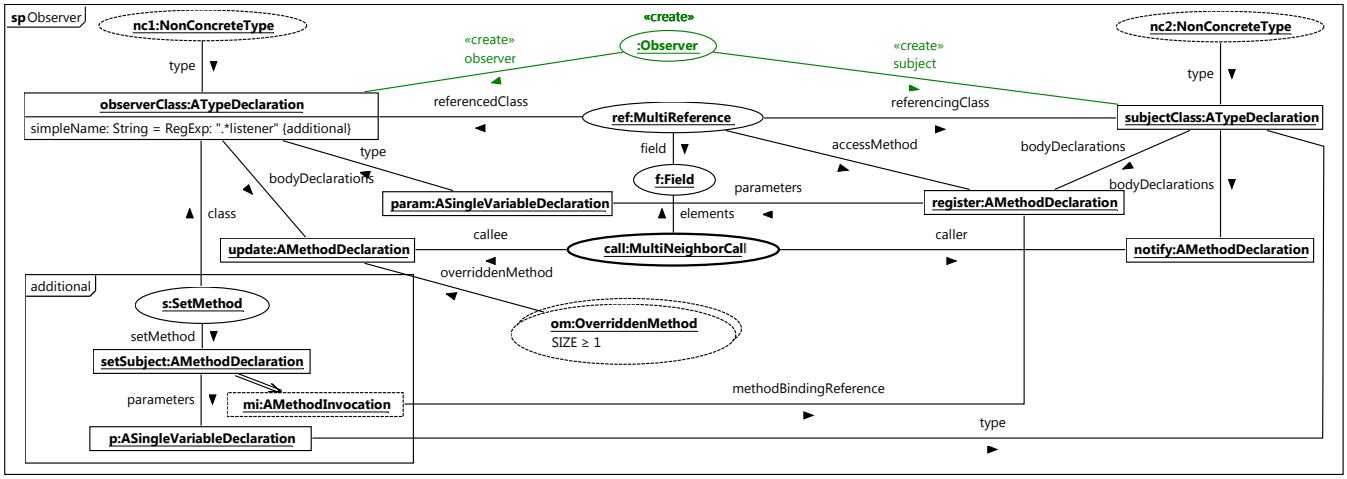


**Figure 1: *Observer* structural pattern**

In the following, the pattern specification and the presentation of the pattern detection results are explained in detail.

## 2.1 Pattern Specification

In Reclipse, a pattern's structure is specified with a pattern specification language based on graph grammars, the so-called *structural patterns*. Throughout this paper, we use the *Observer* pattern as an example. The Observer pattern's intent is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [4]. Figure 1 depicts our specification of the Observer pattern. The rectangular objects represent elements of the system's abstract syntax graph, e.g. classes and methods. The objects are variables that are matched to real objects in the given ASG during pattern detection. The ellipses are so-called *Annotations* and represent subpatterns that are specified in other diagrams. Each element is a condition of the pattern specification.

When the depicted structure is detected in an ASG during the patten detection process, the `Observer` annotation that is marked with `create` is created. It tags the structure as candidate for the Observer pattern and marks objects that play key roles in the pattern (here the observer class and the subject class).

The Observer pattern's structure contains the classes `subjectClass` and `observerClass`. The observer class has a method `update`. The subject class has the methods `register` and `notify`. The `register` method takes an object of the type `observerClass` as parameter. The subpattern `MultiReference` expresses that a subject references arbitrarily many observers. The subpattern `MultiNeighborCall` specifies that the subject's `notify` method contains multiple calls of the observer's `update` method.

The dashed lines (e.g. of the `NonConcreteType` annotations) indicate objects that are not mandatory for the detection of the Observer pattern. They form additional conditions. In the same way, the subgraph in the rectangle to the lower left marked with `additional` (a so-called *additional fragment*) is not mandatory for the detection of an Observer pattern implementation. Detected additional elements increase the number of satisfied conditions and thus the candidate's rating value. The `observerClass` element includes an additional condition on its `simpleName` attribute that defines a condition for the name of the type bound to the observer role. The name has to match the specified regular expression which, in this case, declares that the string should end with "listener".

The `OverriddenMethod` annotation `om` is drawn with a second border and thereby marks the node to represent a set of objects in the ASG. This means that an arbitrarily large number of objects can be mapped to this element. The expression "SIZE $\geq$ 1" indicates that there has to be at least one element in this set.

More details on the specification language used in Reclipse can be found in other publications [5, 7, 10].

## 2.2 Pattern Detection Results View

Figure 2 shows the current results view of Reclipse. It presents an excerpt of the detection results of a static pattern detection on JHotDraw 5.1 [10]. Besides others, we found some candidates for the Observer pattern. The view lists one candidate with its annotated elements (i.e. `observerClass` and `subjectClass`) and additionally shows the rating value and the detected subpatterns with their ratings. In this example, Reclipse detected an Observer pattern candidate with a class named `StandardDrawing` that plays the role of the subject class and a class `DrawingChangeListener` that represents the observer class. All subpatterns (`Field`, `InterfaceType`, `MultiNeighborCall`, `MultiReference` and `OverriddenMethod`) were detected with a rating value of one hundred percent. However, the Observer candidate as a whole only received a rating of 79.31%. That means that some of the conditions in the pattern specification are not satisfied. Unfortunately, the user is not able to see where exactly the candidate deviates from the specification, i.e. which conditions of the corresponding pattern are not satisfied. Furthermore, it is not shown which other objects besides observer and subject class were matched. For

| Annotation | Rating | Annotated Elements |
|---|---|---|
| Observer | 79,31% | observer=CH.ifa.draw.framework.DrawingChangeListener, subject=CH.ifa.draw.standard.StandardDrawing |
| detected subpatterns ... | | |
| Field | 100,00% | fragment=fListeners, type=CH.ifa.draw.standard.Vector, declaration=no Name, owningClass=CH.ifa.draw.standard.StandardDrawing |
| InterfaceType | 100,00% | type=CH.ifa.draw.framework.DrawingChangeListener |
| MultiNeighborCall | 100,00% | calleeClass=CH.ifa.draw.framework.DrawingChangeListener, callee=drawingInvalidated, caller=figureInvalidated, callerClass=CH.ifa.draw.standard.StandardDrawing, elements=Field |
| MultiReference | 100,00% | referencedClass=CH.ifa.draw.framework.DrawingChangeListener, referencingClass=CH.ifa.draw.standard.StandardDrawing, field=Field, accessMethod=addDrawingChangeListener |
| OverriddenMethod | 100,00% | subClass=CH.ifa.draw.standard.StandardDrawingView, overriddenMethod=drawingInvalidated, overridingMethod=drawingInvalidated, superClass=CH.ifa.draw.framework.DrawingChangeListener |

**Figure 2: Detection results from a static pattern detection of JHotDraw 5.1**

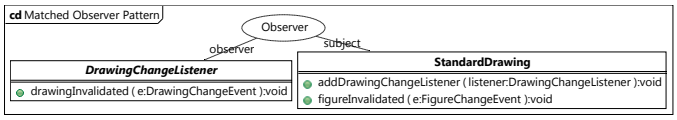


**Figure 3: The class view of the candidate**

example, the user cannot see which concrete methods were matched for the register, the notify, or the update method roles from the specification.

Essentially, the user has no overview of the (un-)satisfied conditions. As a result, it is not comprehensible why the actual rating values are as they are and the detection results can barely be interpreted.

# 3. VISUALIZATION REQUIREMENTS

Visualization is important for pattern detection tools because it helps to envision the detection results so that the user can easily understand the system [3]. Backofen identified several requirements for an adequate visualization of detection results of a static pattern detection [1]:

**R1** To attain clarity and comprehensibility, a compromise between a detailed visualization and a compact, well-arranged view has to be found.

**R2** The presentation of the detection results should show which conditions of a pattern specification are satisfied and which are not.

**R3** To provide a better understanding of the pattern detection results, it should be easy to relate the matched pattern candidate to the pattern's specification.

**R4** All specification elements that were matched to the pattern candidate should be visualized accordingly to provide the user with detailed information about a candidate.

**R5** The concrete values of an object's attributes should be presented to inform the user about the concrete reason why a condition is (not) satisfied.

# 4. PATTERN MATCHING VIEWS

As an addition to the current presentation of detection results as a list of candidates, we developed a graphical visualization, the *pattern matching views*. In the following the new visualization approach is presented.

The Reclipse tool suite, which is based on Fujaba, is a collection of plug-ins for Eclipse. The pattern matching views were also realized in an Eclipse plug-in. There are three different views: The *class view*, the *pattern view* and the *abstract syntax view*. The views can be displayed for each detected pattern candidate and satisfy the requirements identified in Section 3.

In the following, the three views are described in detail. As an example, the Observer candidate from Figure 2 is used.

## 4.1 Class View

In Reclipse, we mostly deal with patterns at the design level which are primarily concerned with classes. Accordingly, their natural syntax is a class diagram. Because of this, the *class view* shows the pattern candidate in a UML class diagram. Class diagrams are a language that most users are familiar with, so they can see immediately which classes play the key roles in the candidate. In addition, this illustration is very compact and thereby provides a convenient overview to the user (cf. requirement R1).

Figure 3 shows the class view for the Observer candidate. The `DrawingChangeListener` and the `StandardDrawing` classes from the example in Figure 2 are presented with their roles in the Observer pattern.

## 4.2 Pattern View

The pattern view shows the pattern specification of a pattern candidate, enhanced by information about which conditions are satisfied by the selected candidate, and which are not (cf. requirement R2). Satisfied conditions of the pattern are shown in black. Conditions that are not satisfied are marked as `unsatisfied` and are visualized in gray.

In Figure 4, the pattern view for the Observer candidate is shown. In this example, the objects in the additional fragment on the lower left are conditions that are not satisfied. That means, the candidate has no set method in the Observer class that takes an object of the subject class as parameter and calls the subject's register method. Also, the attribute expression for the name of the observer class is not satisfied: the class' name does not end with "listener". The `NonConcreteType` annotation on the right side is not matched either, which means that the type which represents the subject is neither abstract nor an interface.

## 4.3 Abstract Syntax View

The abstract syntax view shows the subgraph of the ASG that was matched for the candidate. The advantage is that this is similar to the syntax of the pattern specification, which means that the user is able to easily compare the candidate to the specification (cf. requirement R3).

In Figure 5, the Observer candidate is visualized in the abstract syntax view. Here, all matched objects are presented (cf. requirement R4). For example, the observer class

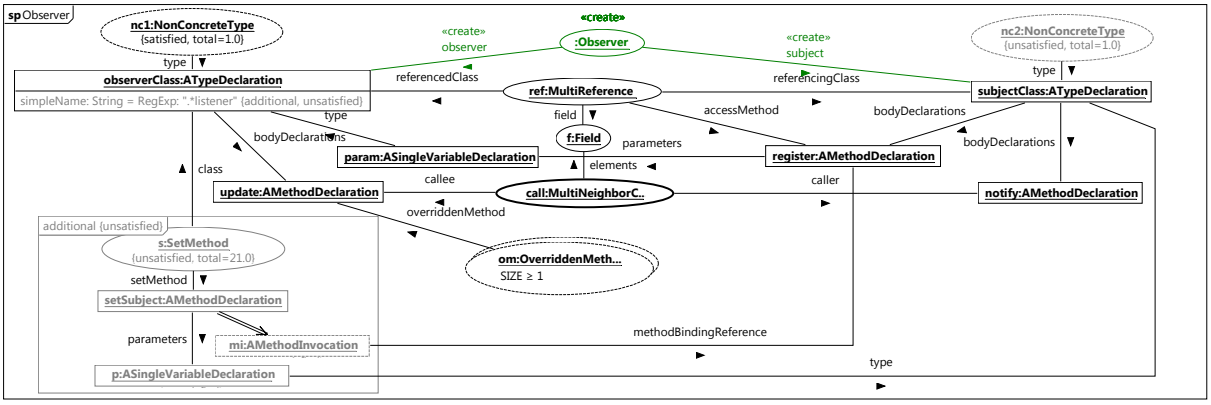


**Figure 4: The pattern view of the candidate**

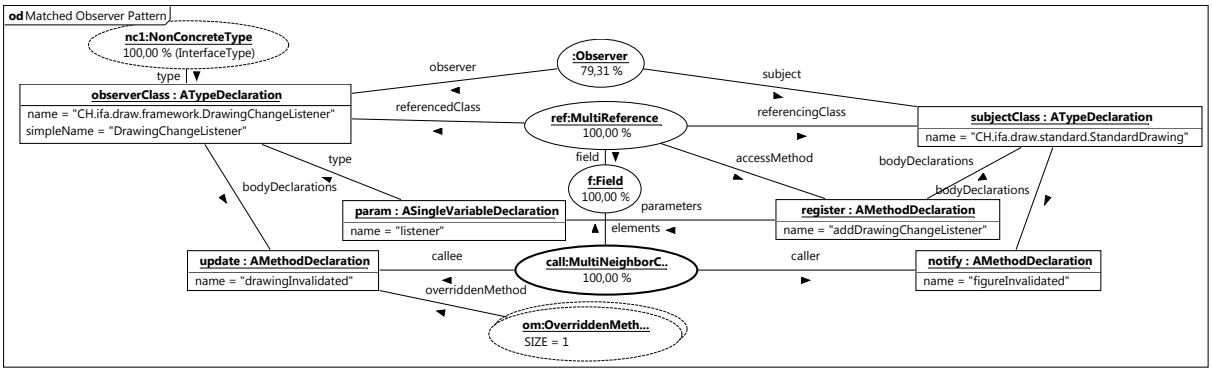


**Figure 5: Abstract syntax view of the candidate**

is named `DrawingChangeListener` and the subject class is named `StandardDrawing`. Also the names of all other matched objects are visualized (cf. requirement R5). The annotations show that the matched subpatterns are all rated with one hundred percent. Additionally, the size of the `OverriddenMethod` annotation is presented. In this example, the update method is overridden only once as indicated by the expressions `SIZE = 1`.

Figure 4 shows that the attribute expression of the observer class' name is not satisfied. The reason for this is revealed in the abstract syntax view: The name of the class is `DrawingChangeListener`. This does not match the regular expression ".*listener" from Figure 4. This hints at a flaw in the specification. The expression could be corrected to ".*(l|L)istener" to improve this condition.

## 4.4 Additional Features

To support the user in comparing the pattern candidate and the specification, the three views provide a consistent selection. If an element in one of the views is selected, the corresponding elements in the other views are highlighted as well.

To simplify the comparison, the layout of the elements shown in the pattern view and in the abstract syntax view is based on the layout of the pattern specification (cf. requirement R3). The user is able to customize the layout of all three views by dragging the elements to new positions.

To enable a clear, well-arranged view of the pattern candidate in abstract syntax, only attributes that have a corresponding condition in the pattern specification are shown (cf. requirement R1).

If the selected pattern candidate includes annotations that represent subpatterns, the user is able to directly open the matching views for the subpattern out of the currently opened views. For example, from the visualized Observer candidate in the abstract syntax view, the user can jump to the detected candidate of the MultiReference pattern to see details about the relation between the observer class and the subject class.

Furthermore, if the pattern specification contains sets of objects or annotations, the user can expand the contained elements for inspection by selecting an action from the context menu. In the pattern view for the Observer candidate, for example, the user can display all methods that are bound to the OverriddenMethod annotation, i.e. all methods that override the observer's update method.

# 5. RELATED WORK

There are many approaches which deal with the detection of patterns. In their survey paper, Dong et al. present several pattern detection approaches that also provide visualization support [3]. Most of those tools present their results as UML class diagrams, in which the pattern roles are marked. One of the approaches proposes a UML profile containing new stereotypes, tagged values and conditions and thereby ex-

tends UML diagrams for visualizing pattern-related issues [2]. Another visualization technique proposed by Dong et al. is a class hierarchy in addition to class diagrams. There, the first level nodes under the root are the classes participating in the pattern while the roles that a class plays are defined as their children [3].

Wiebe et al. use a pattern detection approach similar to the analysis Reclipse uses [12]. After executing a graph matching algorithm, the detected pattern candidates are evaluated and presented. However, the candidates are visualized exclusively as UML class diagram.

Schauer and Keller present an approach where the pattern candidate is juxtaposed with the description from literature [6]. But the informal description is not equivalent to the used formal pattern specification. Thus this approach is not sufficient because it does not provide appropriate information about the discrepancies between pattern specification and candidate.

In summary, none of these pattern detection tools satisfies all of the requirements described in Section 3.

# 6. CONCLUSIONS AND FUTURE WORK

With the pattern matching views, Reclipse provides a visualization of pattern candidates that illustrates the detected candidates in a comprehensive and comprehensible way. Our visualization results in a more transparent rating and thereby supports the user by simplifying the decision if a candidate is a false positive or a real pattern implementation. Furthermore, the user now can compare pattern candidates to the specification. In our Observer example, we received a more detailed view of the classes in JHotDraw which are responsible for updating a drawing because the matching views displayed the methods that play important roles in this mechanism. Furthermore, we were able to correct our Observer specification, because we noticed the flawed attribute condition for the observer class name.

However, the visualization approach still provides space for enhancements. For instance, only attributes that have a corresponding condition in the pattern specification are shown, which is useful, but in some cases not sufficient. The user should be given the additional possibility to view the values of attributes which are not involved in the pattern specification to get a more detailed view of the candidate. An additional idea for the visualization of a candidate is a source code view. Another interesting feature could be the comparison between several candidates of the same pattern.

Moreover, the detection results are non-persistent at the moment. The ability to save the results would allow the user to review them later and to compare different results from multiple analysis runs. This would further support the flaw detection in pattern specifications.

Furthermore, Reclipse also provides a dynamic analysis that analyzes a pattern candidate's runtime behavior. The dynamic pattern detection can be used to reject or verify pattern candidates from the static analysis based on their behavior [9, 11]. The results of the dynamic pattern detection could be used to enhance the pattern matching views by additional information. Thereby the user could gain an even more comprehensive illustration of the detected design pattern implementations in the analyzed software.

# 7. ACKNOWLEDGMENTS

We would like to thank Andre Backofen for his conceptual work [1] on the approach and his help in implementing the pattern matching views in Reclipse.

# 8. REFERENCES

[1] A. Backofen. Visualisierung von Musterfunden bei der statischen Software-Muster-Erkennung. Bachelor's thesis, University of Paderborn, Nov. 2009.

[2] J. Dong, S. Yang, and K. Zhang. Visualizing design patterns in their applications and compositions. *IEEE Transactions on Software Engineering*, pages 433–453, 2007.

[3] J. Dong, Y. Zhao, and T. Peng. A Review of Design Pattern Mining Techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 2009.

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

[5] 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 (ICSE), Orlando, FL, USA*, pages 338–348. ACM Press, May 2002.

[6] R. Schauer and R. Keller. Pattern visualization for Software Comprehension. In *Proceedings of the 6th International Workshop on Program Comprehension*, pages 4–12. IEEE, 2002.

[7] M. von Detten, M. Meyer, and D. Travkin. Reclipse – a reverse engineering tool suite. Technical Report tr-ri-10-312, University of Paderborn, Paderborn, Germany, 2010.

[8] M. von Detten, M. Meyer, and D. Travkin. Reverse Engineering with the Reclipse Tool Suite. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, May 2-8, 2010*, volume 2, pages 299–300. ACM Press, May 2010. Informal Research Demonstration.

[9] M. von Detten and M. C. Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proceedings of the 7th International Fujaba Days*, pages 15–19. Eindhoven University of Technology, 2009.

[10] M. von Detten and D. Travkin. An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw. Technical Report tr-ri-10-322, Universitiy of Paderborn, 2010. Vers. 1.0.

[11] L. Wendehals. *Struktur- und verhaltensbasierte Entwurfsmustererkennung*. PhD thesis, University of Paderborn, September 2007. In German.

[12] E. Wiebe, S. Keul, S. Staiger, and G. Vogel. Entwurfsmuster-erkennung mit bauhaus. In *Proceedings of the 10th Workshop Software Reengineering*, volume 126 of *LNI*, pages 181–185. GI, 2008.