An Evaluation of the Reclipse Tool Suite based on the Static Analysis of JHotDraw

Technical Report tr-ri-10-322

Markus von Detten, Dietrich Travkin

Software Engineering Research Group
Heinz Nixdorf Institute, Department of Computer Science
University of Paderborn, Paderborn, Germany
[mvdetten|travkin]@upb.de

Version 1.0

Abstract. The detection of software design pattern implementations in existing code helps reverse engineers to understand the software design and the original developers' intentions. In order to automate the tedious and time-consuming task of finding pattern implementations several research groups developed pattern detection algorithms and reported their precision and recall values to compare the approaches.
Our research group developed another approach for the detection of software patterns, Reclipse, that exhibits some unique features like fuzzy expressions to better describe patterns and rate the detected pattern occurrences. For evaluation, we applied our pattern detection approach to JHotDraw. In the following, we present and compare our pattern detection results with those of other approaches.

1 Introduction

During their life cycle software systems have to be adapted to new requirements, new features have to be added, defects have to be eliminated. These tasks often involve the adaption of the software design and require knowledge of the current design. Unfortunately, a system's design documentation is often not up-to-date or missing. Thus, before performing any changes to the system, developers usually have to recover the system design, which is known as *reverse engineering* [4].

Software design patterns (e.g. [1–3, 5, 7, 10, 11, 16]) are approved solutions to recurring software design problems. They are documented by experts and increase flexibility and maintainability by introducing extensible or adaptable software design so that it is easier to add new features to or adapt software systems. In order to make such patterns applicable in many situations, independently from the programming language or the chosen platform, their descriptions are kept informal and very general. The most famous and widely used collection of patterns are the *Design Patterns* [7] described by the so called *gang of four (GoF)*: Gamma, Helm, Johnson, and Vlissides.

Since patterns describe the system design and are used to achieve a certain, well documented goal, the detection of pattern implementations in code can help to understand the system design and provides hints on the original developers' intentions.

The manual inspection of code for the purpose of the detection of pattern implementations is a very tedious and time-consuming task. Several approaches are aiming at automatically detecting implementations of software patterns (e.g. [6, 8, 9, 12, 14, 17, 19, 21]) in order to speed up the detection process. Our research group developed another approach for the detection of pattern implementations. After introduction of the first ideas [14] by Niere et al., we continuously extended and improved our approach [21, 22]. The current implementation of our approach is called *Reclipse*[1] [20, 21].

For the comparison of precision and recall other researchers evaluated their approaches by applying them to software systems for which the design patterns contained therein are well documented. One of the software systems which is most frequently used for such evaluations is JHotDraw[2]. In order to evaluate our detection approach and compare our work with that of other researchers, we used our detection tool, Reclipse, to detect implementations of the gang-of-four design patterns [7] in JHotDraw. In the following, we present our detection results for JHotDraw 5.1 and compare them with those of other approaches.

2 The analyzed system: JHotDraw 5.1

JHotDraw is an application framework that supports the development of customized drawing editors. It has been continuously developed for many years and has currently reached version 7.5.1. Because most related work focuses on lower versions of JHotDraw, we used version 5.1 for our evaluation.

JHotDraw is written in Java. The analyzed version 5.1 consists of 155 classes and a total of 7993 lines of code (excluding comments).

3 Related Work

We compared our detection results with six other publications and two additional resources. Table 1 shows the related work we chose for this comparison. It presents the authors and titles of the publications, the year of publication, the short hand reference which we use throughout this report to refer to the publication, and the version of JUnit which was used for the evaluation of a given approach.

The related approaches can be also be distinguished by their level of detail regarding the detected pattern occurrences. While most publications only present the number of detected occurrences for each pattern, some of them explicitly state which classes constituted which pattern occurrence. This is reflected by

[1] http://www.fujaba.de/reclipse
[2] http://www.jhotdraw.org

the final column *Detailed Results*. The approaches which report those detailed results are also referred to in Section 4 in which we present our detected pattern occurrences.

#	Title	Year	Reference	Version	Detailed Results
1	J. Dong, Y. Zhao, Y. Sun : Design Pattern Detection by Template Matching	2008	DSZ08 [6]	6.0	No
2	Y.-G. Guéhéneuc, G. Antoniol : DeMIMA: A Multi-Layered Approach for Design Pattern Identification	2008	GA08[8]	5.1	No
3	Y.-G. Guéhéneuc, H. Sahraoui, F. Zaidi : Fingerprinting Design Patterns	2004	GSZ04 [9]	5.1	No
4	A. De Lucia, V. Deufemia, C. Gravino, M. Risi : Improving Behavioral Design Pattern Detection through Model Checking	2010	LDGR10 [12]	5.1	Yes
5	N.Shi, R. Olsson : Reverse Engineering of Design Patterns from Java Source Code	2006	SO06 [17]	6.0	No
6	N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis : Design Pattern Detection Using Similarity Scoring	2006	TCSH06 [19]	5.1	Yes
7	D.Riehle : Framework Design: A Role Modeling Approach	2000	Rie00 [15]	5.1	Yes
8	JHotDraw 5.1 Javadoc	2001	Javadoc	5.1	Yes

Table 1. Related publications to which we compared our detection results

Table 2 shows a comparison of the different detection results. In the first column the a selection of detected patterns is shown[3]. The other columns present the number of detected pattern occurrences for Reclipse and each of the six other approaches. Note that '–' does not signify that a pattern was not found but that the pattern in question was not considered at all by an approach.

It is noteworthy that the number of detected pattern occurrences varies widely between the different approaches. While the analysis of Guéhéneuc et al. [9] (using Fingerprinting) yielded relatively few results, Guéhéneuc and Antoniol [8] (with the DeMIMA approach) found significantly more pattern occurrences.

Tsantalis et al. [19] claim that their approach did only detect true positives and no false positives. A close look at the detailed results[4], however, puts this statement into question. Many of the results that we detected and, after a manual inspection, labeled as false positives are reported as true positives by Tsantalis et al.

[3] Reclipse and also some of the other approaches are capable of detected more patterns than the ones considered in this report. For example, we also used Reclipse to detect additional Design Patterns (e.g. Bridge, Chain of Responsibility), Anti Patterns [13], and guideline violations in MATLAB/Simulink specifications [18].

[4] http://java.uom.gr/∼nikos/pattern-detection.html

Pattern	Reclipse[1]	DSZ08	GA08[1]	GSZ04	LDGR10	SO06[2]	TCSH06[3]
Adapter[4]	67	51	28	1	–	≈ 2	18
Command	5	–	11	1	1	–	18
Composite	1	0	3	1	–	≈ 1	1
Decorator	2	1	13	1	–	≈ 2	3
Observer	5	–	7	2	9	≈ 5	5
Singleton	2	–	2	2	–	≈ 1	2
State	16	29	21	2	36	≈ 1	22
Strategy	16	–	21	4	43	≈ 50	22
Template Method	18	–	31	2	1	≈ 1	13

Table 2. Related publications to which we compared our detection results

[1] The approach does not distinguish State/Strategy patten occurrences.
[2] The authors provide only a bar chart to represent their results and report no exact figures.
[3] The approach does not distinguish State/Strategy and Adapter/Command pattern occurrences, respectively.
[4] As JHotDraw is a Java system, all mentions of the Adapter pattern refer to the Object Adapter pattern variant from [7] (as opposed to the Class Adapter pattern, which relies on multiple inheritance).

Regarding the number of detected results, Reclipse lies somewhere in the middle but in contrast to the listed approaches, Reclipse also offers a rating of the detected occurrences to represent a candidate's adherence to the pattern specification. These ratings are shown and discussed in Section 4.

J. Dong, Y. Zhao, Y. Sun : Design Pattern Detection by Template Matching [DSZ08]

Dong et al. [6] apply a graph comparison approach similar to Tsantalis et al.[19], but instead of calculating node similarities they determine the overall similarity of a pattern candidate graph to the corresponding pattern specification graph. The similarity score is used to rate the candidates quality. Nevertheless, there is no possibility to specify gradually satisfiable constraints comparable to the fuzzy constraints used in our specifications. Furthermore, similar to Tsantalis et al., the adjacency-matrix-based pattern specifications only cover structural properties. Behavior remains unconsidered and complex software metrics cannot be used.

Y.-G. Guéhéneuc, G. Antoniol : DeMIMA: A Multi-Layered Approach for Design Pattern Identification [GA08]

In the DeMIMA pattern detection approach [8] patterns are specified by a set of constraints describing class relations and software metric ranges. If all constraints are satisfied by a pattern candidate, an exact matching is reached. Deviated pattern instances are detected by (automatically or manually) relaxing

the specified constraints whereby an approximate matching is achieved. Similar to our approach, the weighted number of relaxed constraints is used to calculate a percentage rating describing the similarity between the pattern specification and the pattern candidate. Although this is a promising approach, instead of interactively relaxing constraints, which is time consuming, in our approach, the mandatory and relaxable constraints are made explicit in the pattern specifications. Furthermore, in contrast to DeMIMA, we support gradual metric constraint satisfaction, thereby avoiding inappropriate thresholds, and providing a more precise candidate rating.

Y.-G. Guéhéneuc, H. Sahraoui, F. Zaidi : Fingerprinting Design Patterns [GSZ04]

Guéhéneuc et al. [9] propose to characterize design patterns by a combination of metric values and thereby obtain "fingerprints" for those patterns which can be used to identify occurrences of them. However to create the fingerprints, a machine learning algorithm is used which requires a repository of pattern examples. The authors evaluated this idea on multiple software systems, among them JHotDraw 5.1.

A. De Lucia, V. Deufemia, C. Gravino, M. Risi : Improving Behavioral Design Pattern Detection through Model Checking [LDGR10]

De Lucia et al. [12] improve their static analysis with dynamic analysis approaches. Before carrying out the dynamic analysis however, they use a model checker to sort out false positives that are statically recognizable. For their evaluation, they also used JHotDraw 5.1 and compared their results to other approaches. They also provide detailed discussions for some concrete pattern occurrences.

N.Shi, R. Olsson : Reverse Engineering of Design Patterns from Java Source Code [SO06]

Shi and Olsson present a static pattern detection approach and compare it to two other approaches, one of them being FUJABA which was the prototype implementation of our research group's initial pattern detection approach [14] and Reclipse's precursor. They focus on the ability to detect patterns at all and not so much on single occurrences or the concrete results. Thus, they only provide a small bar chart to present their detection results. This makes it hard to compare their results to ours.

N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, S.T. Halkidis : Design Pattern Detection Using Similarity Scoring [TCSH06]

Tsantalis et al. [19] compare a graph representing a candidate with a graph representing a pattern by calculating the similarities of the contained nodes.

The most similar nodes are chosen to build pattern candidates. The node similarities could be combined to determine the similarity of a candidate and the corresponding pattern and thereby rate the candidate. Unfortunately, no rating is provided and the node similarities are not reported to the reverse engineer. Furthermore, Tsantalis et al. only consider classes and corresponding relations in their pattern specifications while we also take the behavior into account by covering statements in method bodies.

They carried out their evaluation on JHotDraw 5.1 and provided detailed results on an additional web site (http://java.uom.gr/∼nikos/pattern-detection.html) which complements the evaluation in the paper. Due to structural similarities of the patterns the authors do not distinguish between Adapter and Command pattern occurrences and State and Strategy occurrences, respectively.

Furthermore, the authors claim that their approach did only detect true positives and no false positives. After comparing their results to ours, we (in accordance with [12]) do not share that view.

Additional Resources

For a detailed comparison of the detected pattern occurrences, we used two additional resources. The dissertation of Dirk Riehle and the JHotDraw Javadoc documentation both provide detailed information about actual pattern implementations in JHotDraw. In Section 4 these resources are referenced by *Rie00* and *Javadoc*, respectively.

D.Riehle : Framework Design: A Role Modeling Approach [Rie00] In his dissertation [15], Dirk Riehle presents a role model-based approach for framework development. For its evaluation, he carried out a case study on JHotDraw which provides extensive information on the contained pattern occurrences in the JHotDraw system. In contrast to the other referenced approaches, Riehle did not use an automatic pattern detection tool.

JHotDraw 5.1 Javadoc The Javadoc documentation of JHotDraw explicitly mentions some of the patterns that were employed during the design of the software. We used this resource to validate pattern occurrences detected by Reclipse.

4 Detected Patterns

For each of the following patterns we first show our specification and explain it briefly. Second, we provide a table which details our detection results for that pattern and explain our findings. For those patterns for which a lot of occurrences were detected by Reclipse, we provide an evaluation of a sample of the most promising (i.e. highest rated) candidates.

The Reclipse pattern specifications are essentially object diagrams that represent an exemplary object structure which has to be detected for a given candidate. Apart from objects and links it also contains subpatterns (black, elliptical

nodes) which represent structures that occur in many patterns and are therefore reusable in lots of specification [5]. Their presence is a prerequisite for a pattern that contains them to be found. Each specification also contains one green, elliptical annotation. This annotation is created if an object structure which matches the specification is detected. For more information on the pattern detection process, we refer the reader to our previous publications [14] [20] [21].

The tables shows the detected pattern occurrences with their ratings and specifies which classes and methods play characteristic roles in the pattern candidate (such as subject and observer in the Observer pattern). In the column *Documented in*, we list the related publications which report the same pattern occurrence. If the authors declared a specific candidate to be a false positive, this is marked by *(FP)*. The column *Evaluation* states our own assessment of the candidate: Is it a true or false positive?

4.1 Adapter

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because incompatible interfaces." [7]

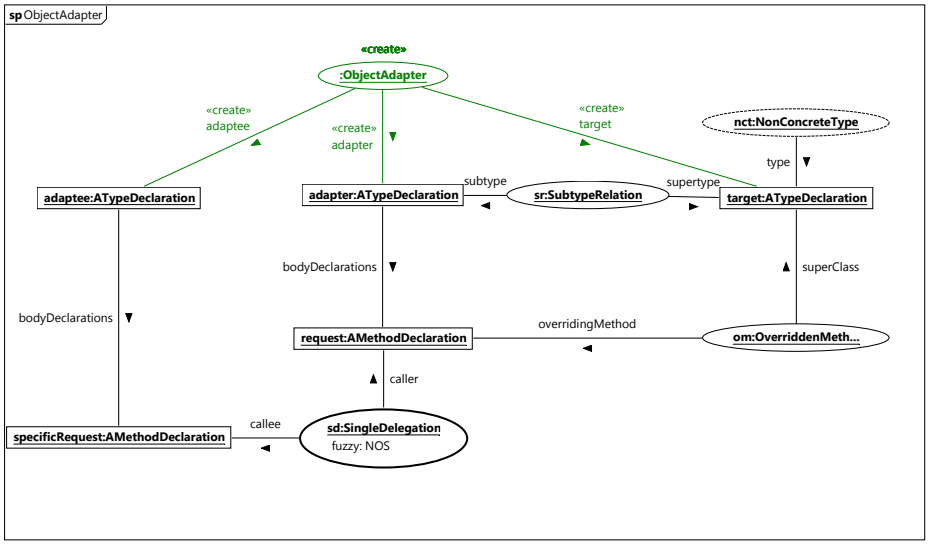

Fig. 1. Reclipse Specification of the Adapter Pattern

[5] On example is the overridden method subpattern which occurs in our specifications of the Adapter, Command, Decorator, Observer, State, Strategy, and Template Method patterns. The subpattern specifications can be found in the appendix.

Specification Our specification of the adapter pattern contains three types: the adaptee, the target and the adapter which is responsible for mediating between those two. The adaptee has a specific request method which represents the legacy interface. The adapter class is a subclass of the target class and overrides the request method of its super type. This overridden request method simply delegates the request to the adaptee's specific request method. This is represented by the SingleDelegation subpattern.

We added a Fuzzy Metric Constraint to the single delegation because a correctly implemented delegation should consist of a delegating statement and (nearly) nothing else. Consequently, we used the following function to rate the number of statements in the delegation.

$$\mu(x) = \frac{1+\varepsilon}{1+e^{\frac{x}{compr}}} \text{ with } \varepsilon = 1, \text{compr} = 4$$

Fig. 2. Fuzzy function for the number of statements in the Adapter pattern's delegation

This yields a strictly decreasing, exponential function which comes close to its limit 0 for $x \approx 16$ (cf. Figure 2). This means that delegations are rated worse, the more statements they have, nearing the rating 0 at about 16 statements.

Detection Results It is noteworthy that the structural aspects of the Adapter pattern are not very unique. The pattern essentially describes a delegation between two classes (Adapter and Adaptee) where one of those classes also has a super class (Target). This structure appears frequently in implementations, often without the intention to be a dedicated implementation of the Adapter pattern. Thus, the pattern described by Gamma et al. is too unspecific to serve as a uniquely identifiable structural pattern which is also reflected by the large number of detected occurrences for other approaches (cf. Table 2).

For the Adapter pattern, 67 pattern occurrences were detected by Reclipse. Table 3 shows a small sample of those detection results. The occurrences at the

top of the list effectively are true positives albeit without explicit documentation by the developers. One main point that sets the detected occurrences apart, besides the NonConcreteType annotation and the satisfaction of the fuzzy expression, is the rating of the SingleDelegation annotation. The detected delegation for the lower rated patterns contains many statements and the reference used therein often lacks proper access methods. But even the low-ranked occurrences could still be regarded as true positives because they exhibit the core features of the pattern as explained above.

Another thing to note is a problem with the detection of pattern combinations. Consider results #1 and #2 or #8 and #9, respectively. It it obvious that the pattern occurrences are just different combinations of the same classes. The Target role is played by either an interface (Figure, Handle) or an abstract class that implements it (AbstractFigure, AbstractHandle). Otherwise, the occurrences are identical and also receive the same rating.

The occurrences essentially belong to only two Adapter occurrences and just name different classes for the roles which are somehow distributed over an inheritance hierarchy. For occurrences #8 and #9, Rie00 and the JHotDraw Javadoc take this into account and state that the actual Adapter pattern occurrence is between Figure (i.e., the interface which ConnectionFigure implements) and Handle. Considering this problem, many of the detected Adapter pattern occurrences could be merged, so that significantly less occurrences would be reported by Reclipse. A possible specification language extension to achieve this is discussed in Section 5.

#	Rating	Roles			Documented in	Evaluation
		Adapter	Adaptee	Target		
1	96,85%	TextFigure	OffsetLocator	Figure		True Positive
2	96,85%	TextFigure	OffsetLocator	AbstractFigure		True Positive
3	96,85%	AttributeFigure	FigureAttributes	Storable		True Positive
4-7	95,38%-93,47%	...	...	...		
8	93,15%	ConnectionHandle	ConnectionFigure	Handle	Rie00, Javadoc	True Positive
9	93,15%	ConnectionHandle	ConnectionFigure	AbstractHandle	Rie00, Javadoc	True Positive
10-67	92,72%-51,85%	...	...	...		

Table 3: Detection Results for the Adapter pattern

4.2 Command

> "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations." [7]

Specification The Command pattern is one of the more complex specifications. It shows the four types invoker, abstractCommand, concreteCommand and receiver, where concreteCommand obviously is a subtype of abstractCommand as specified by the SubtypeRelation annotation. invoker has two methods, setCommand and invoke; receiver has an action method and both commands have an execute method (to better distinguish the execute methods the abstractCommand's method is called abstractExecute here). Furthermore, the abstractExecute method (and by inheritance also the concrete execute method) must not have a parameter, indicated by the crossed-out param object. invoker.setCommand has a parameter of the type abstractCommand whereas somewhere in the invoker.invoke method there is an invocation of the abstractExecute method. The fact that this invocation is "somewhere" in the invoke method is reflected by the path link (double-lined arrow) between invoke and the method invocation mi2. Similarly, the receiver's action method is called by the command's execute method and the concreteCommand's constructor should be parameterized with the command's receiver. Finally, there should be references from invoker to the abstractCommand and from the concreteCommand to its receiver. References that point the other way round are explicitly prohibited by the two negative (crossed-out) Reference annotations ref_neg1 and ref_neg2.

Detection Results The results for the Command pattern are shown in Table 4. The first two detected occurrences are true positives. However, they might be considered as the same pattern occurrence with different invoker classes. These occurrences are also reported by Tsantalis et al. (although there they cannot be distinguished from the Adapter pattern) and documented in the JHotDraw Javadoc.

Note that the other detected occurrences are just slightly lower rated than the true positives due to structural similarities. This indicates that our specifications should be tuned to better differentiate them from the true positives.

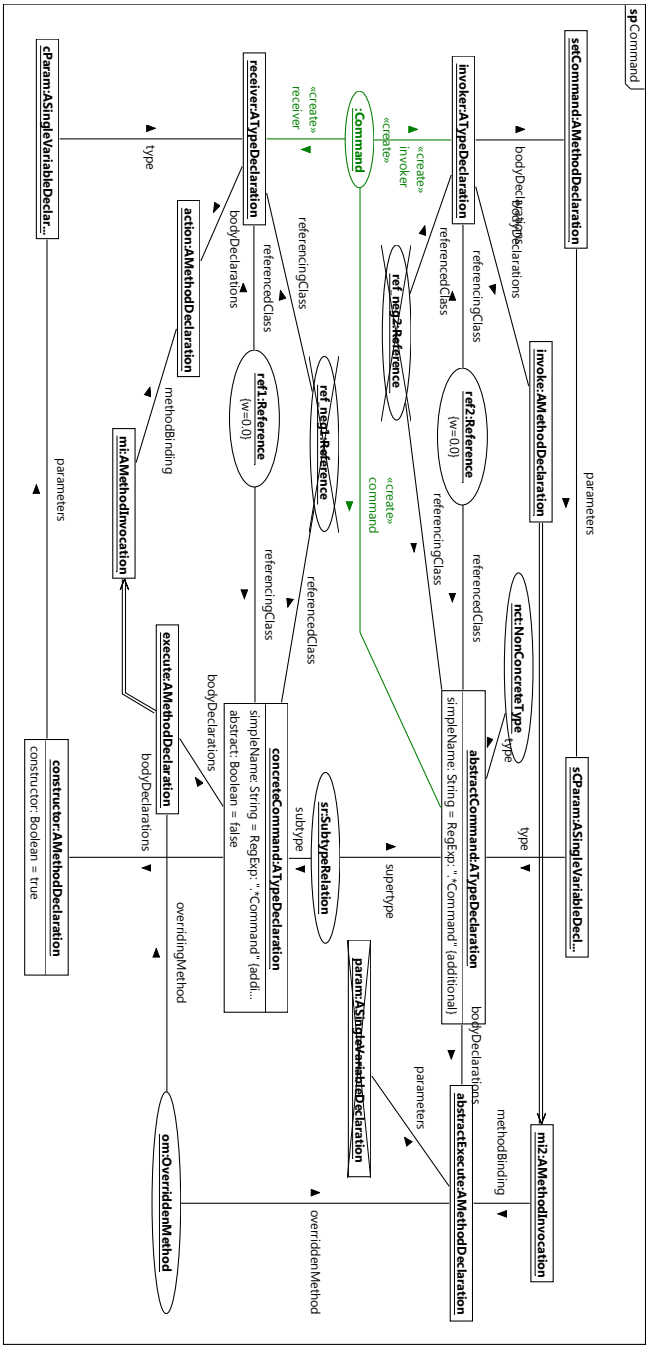

Fig. 3. Reclipse Specification of the Command Pattern

#	**Rating**	**Roles**			**Documented in**	**Evaluation**
		Command	Receiver	Invoker		
1	100%	Command	DrawingView	CommandButton	TCSH06, Javadoc	True Positive
2	100%	Command	DrawingView	CommandMenu	TCSH06, Javadoc	True Positive
3	93,94%	Tool	DrawingView	DrawApplication		False Positive
4	93,94%	Tool	DrawingView	DrawApplet		False Positive
5	93,94%	Connector	Locator	ChangeConnectionHandle		False Positive

Table 4: Detection Results for the Command pattern

4.3 Composite

"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [7]

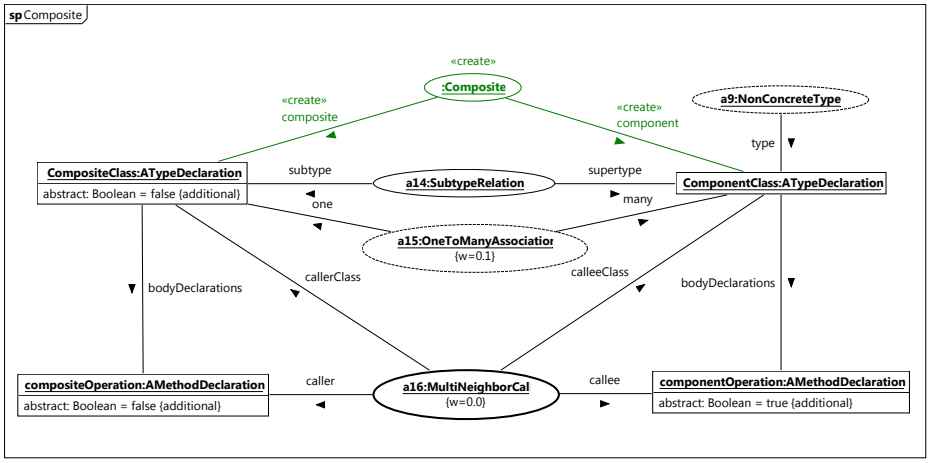

Fig. 4. Reclipse Specification of the Composite Pattern

Specification The Composite specification contains two types, one of which represents the component class and the other the composite class. Between those two a subtype relationship has to exist and, according to [7], the composite class should have a one-to-many reference to its children. Because this reference can be realized in many different ways, we marked it as *additional* in our specification, as shown by the dashed border of the OneToManyAssociation annotation. Both, composite and component should contain a method (compositeOperation and componentOperation, respectively) which, in case of the composite, has to call that same method on all the composite's children. This fact is represented by the MultiNeighborCall between the two methods. The OneToManyAssociation is a very complex subpattern which in turn consists of several subpatterns itself. Thus, the detection of this subpattern means matching over 70 single elements. This leads to the OneToManyAssociation totally dominating the Composite pattern's rating. Thus, we lowered the OneToManyAssociation's weight to 0.1, thereby reducing its impact on the Composite rating.

Detection Results For the Composite pattern one candidate was detected. The rating of the candidate is somewhat low because it does not fulfill all of the specified constraints. According to our specification of the composite pattern a one-to-many association between the composite (*CompositeFigure* in this

case) and its components (*Figure*) is expected. This means that the composite knows all its children and each of those knows its parent in return. The later is not true in the detected pattern occurrence. *Figure* has no to-one reference to *CompositeFigure*. Furthermore the class *CompositeFigure* is abstract which also deviates from our specification. Hence, the rating for this pattern occurrence is rather low.

#	**Rating**	**Roles**		**Documented in**	**Evaluation**
		Composite	Component		
1	45,98%	CompositeFigure	Figure	Rie00, TCSH06	True Positive

Table 5. Detection Results for the Composite pattern

4.4 Decorator

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." [7]

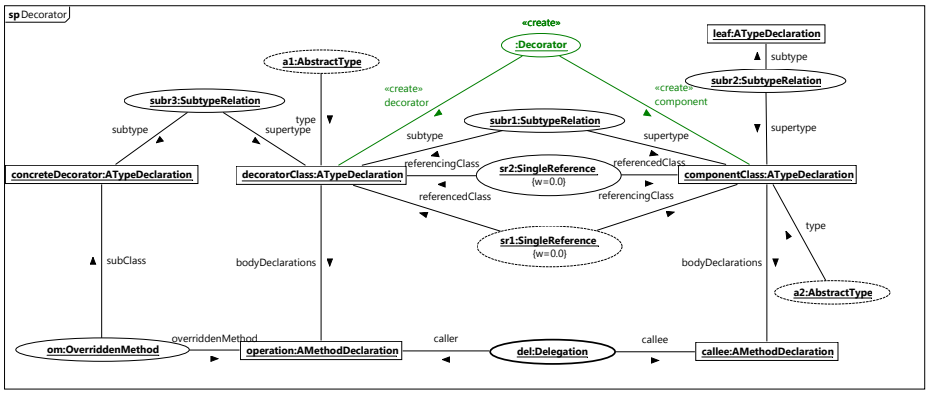

Fig. 5. Reclipse Specification of the Decorator Pattern

Specification Similar to the Composite pattern, this pattern specification contains a component class. It is subclassed by a decoratorClass. There may be more subclasses of the componentClass which are called leaves in the context of this pattern. Subclasses of the decoratorClass are called concreteDecorators. The decoratorClass has an operation which is overridden by the concreteDecorator classes and, on being called, delegates to a method (callee) of the componentClass. In contrast to the composite pattern, there is no one-to-many reference between the decorator and the componentClass but one decorator knows exactly one component (via the SingleReference) and vice-versa. Both, componentClass and decoratorClass should be abstract classes as signified by the AbstractType annotations.

Detection Results We detected two occurrences of the Decorator pattern. The first one, rated with nearly 93% is a true positive and also documented by Riehle, Tsantalis et al., and in the JHotDraw Javadoc. The second occurrence was also reported by Tsantalis et al. [19].It is a false positive that receives a lower rating than the other occurrence for several reasons. The most important points are the absence of a reference from the detected component (Tool) to the decorator class (SelectionTool) and the fact that neither of both classes is abstract.

#	Rating	Roles		Documented in	Evaluation
		Decorator	Component		
1	92,92%	DecoratorFigure	Figure	Rie00, TCSH06, Javadoc	True Positive
2	85,44%	SelectionTool	Tool	TCSH06	False Positive

Table 6. Detection Results for the Decorator pattern

4.5 Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [7]

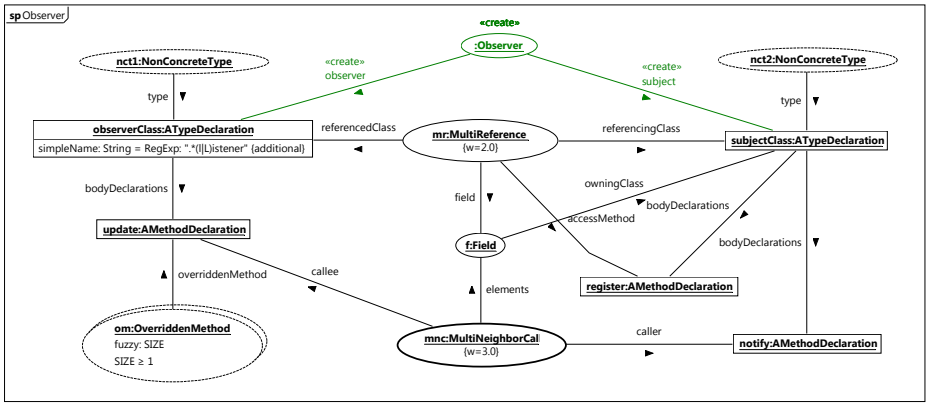

Fig. 6. Reclipse Specification of the Observer Pattern

Specification The Observer pattern consists of the two types observerClass and subjectClass. Both are usually abstract classes or interfaces which is represented by the two NonConcreteType annotations. Because one subject can have multiple observers, there is a MultiReference between them. The observerClass has an update method while the subjectClass has a notify method and a register. The register method accesses the collection which represents the MultiReference between subjectClass and observerClass to register new observers with the subject. The update method is responsible for notifying all observers so there has to be a MultiNeighborCall between update and notify which calls the method on all elements that are accessible via the MultiReference.

The Observer pattern usually is used to allow multiple different observer classes to observe a given subject class. The pattern specification describes the abstract observer class which should then be subclassed by many concrete observers. This is reflected in the specification by the set annotation Overridden-

Method. The update method has to be overridden at least once (specified by the constraint $SIZE \geq 1$ in the OverriddenMethod annotation). To rate those pattern occurrences higher which have more concrete observer implementations, we also added a fuzzy set rating function to the OverriddenMethod annotation. This function is a strictly increasing exponential function with limit 1 and can be described by the formula

$$\mu(x) = \frac{1}{1+e^{(\frac{-x+\Delta X}{compr})}}, with \Delta X = 5, compr = 2$$

The resulting function is depicted in Figure 7. It converges to a rating of one for about 10 overriding methods.

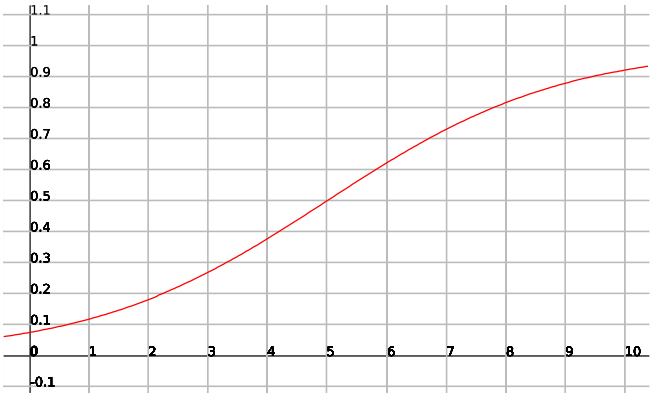

Fig. 7. Fuzzy set rating function for the size rating of the Overridden Method annotation

Detection Results For the Observer pattern five occurrences were found in the code. All these occurrences were also reported by Tsantalis et al. as well as de Lucia et al. Two additional candidates (6 and 7) that were reported in related work were not detected by Reclipse and seem to be not real implementations of the Observer pattern, after all.

Occurrence #1 received the highest rating but nevertheless appears to be a false positive. Its structure is a nearly perfect match of our specification but the method which is detected as the Observer's update method is only a getter (`isExecutable()`) in reality. A dynamic analysis with Reclipse would probably reveal this candidate to be a false positive without further manual examination (cf. Section 6).

Occurrence #2 possesses the same structural features as candidate #1 (thus fulfilling the same constraints) but receives a slightly lower rating. This is because the methods detected to play the role of the update method for the two candi-

dates is overridden a different number of times. For candidate #1 the method is overridden five times, while it is only overridden once for candidate #2.

Occurrence #3's rating is significantly worse than the rating of the first two occurrences because the supposed notify method accesses another field than the supposed register method. This strongly suggests that the occurrence is a false positive.

The fourth occurrence receives a mediocre rating, because it does not iterate over the collection of "Observers" in the alleged notify method.

Occurrence #5 seems to be part of the mechanism of occurrence #2.

Three more false positives were detected as Observer occurrences. They have also been reported by others and their structure is to some extent similar to the Observer pattern. Further explanations are given in the Comments column of the table. The table also lists two more false positives which were detected by other approaches but not by Reclipse.

The Observer pattern, being a pattern with a very distinctive behavior, lends itself well to an additional behavioral analysis to separate true from false positives. Reclipse possesses this capability which has been discussed and evaluated in other publications [20] [22] [23] but is not the focus of this report.

#	Rating	Roles		Documented in	Evaluation
		Subject	Observer		
1	96,96%	CommandMenu	Command	LDGR10 (*FP*), TCSH06 (*FP*)	False Positive
2	91,33%	StandardDrawing	DrawingChangeListener	Rie00, Javadoc, LDGR10, TCSH06	True Positive
3	63,18%	StandardDrawingView	Painter	LDGR10 (*FP*), TCSH06 (*FP*)	False Positive
4	61,52%	CompositeFigure	Figure	LDGR10 (*FP*), TCSH06 (*FP*)	False Positive
5	61,52%	StandardDrawingView	Figure	LDGR10 (*FP*), TCSH06 (*FP*), TCSH06-Web	False Positive
6	– (not found)	Figure	FigureChangeListener	LDGR10	True Negative
7	– (not found)	Connector	ConnectionFigure	Javadoc, LDGR10[6]	True Negative

Table 7: Detection Results for the Observer pattern

[6] reported as ConnectorFigure

4.6 Singleton

"Ensure a class only has one instance, and provide a global point of access to it." [7]

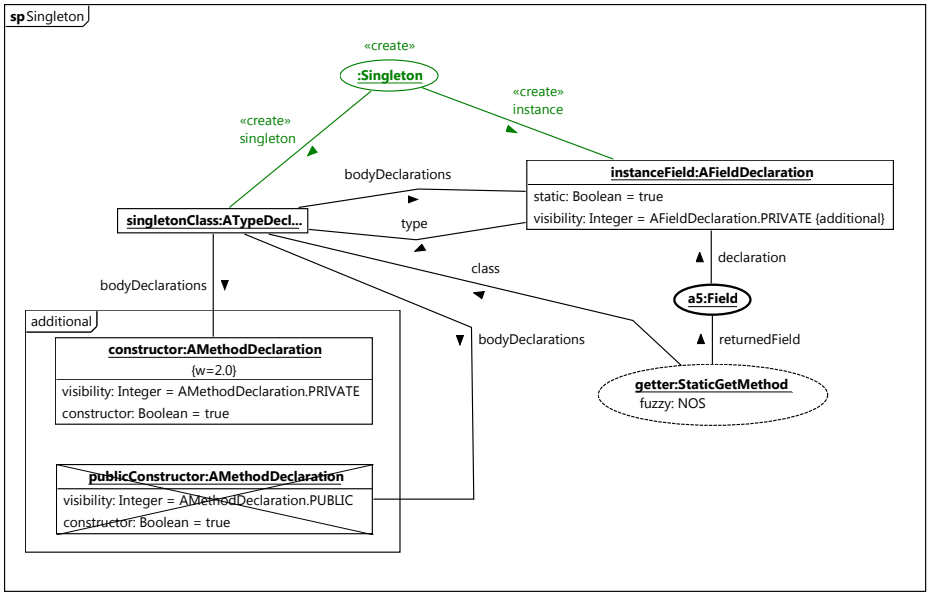

Fig. 8. Reclipse Specification of the Singleton Pattern

Specification The singleton pattern describes a singletonClass which has a static instanceField of the that exact same type. The instanceField's visibility should be private (marked as additional because, based on our experience, this is often violated in implementations). The singletonClass should also contain a static get method through which the instanceField can be accessed. Additionally, the singletonClass's constructor should be private, which is specified by the corresponding method declaration in the additional fragment.

We imposed a Fuzzy Metric Constraint on the get method to reflect the fact that get methods should have few lines of code. We used basically the same function as for the fuzzy metric expressions for the Adapter and Command patterns but changed the parameter *compr* to 2. This makes the functions converge earlier and therefor requires the set method to have even fewer statements than the delegations in the Adapter and Command patterns.

$$\mu(x) = \frac{1+\varepsilon}{1+e^{\frac{x}{compr}}} \text{ with } \varepsilon = 1, \text{compr} = 2$$

This yields a strictly decreasing, exponential function which reaches its limit 0 for $x \approx 8$ (cf. Figure 9). This means that methods which have been identified

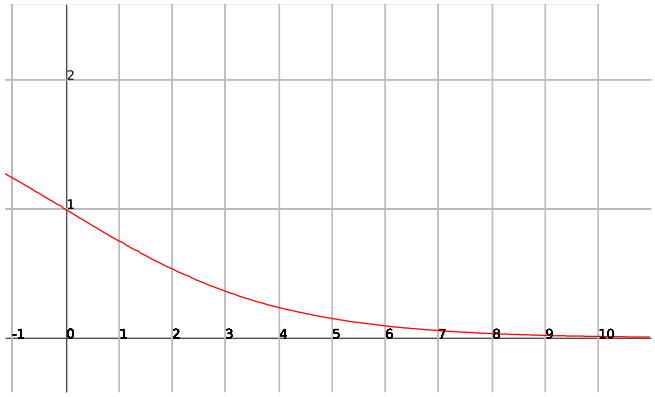

Fig. 9. Fuzzy function for the number of statements in the Singleton pattern's get method

as get methods are rated worse, the more statements they have, converging to rating 0 at about 8 statements.

Detection Results Both Singleton occurrences that were detected are true positives but deviate slightly from our specification which is why their ratings lie below 100%. Clipboard adheres very much to the specification with the only exception of the visibility of the attribute which holds the Singleton instance. The specification states the desired visibilty as *private* whereas the actual visibilty is *package*.

In case of Iconkit, the rating is even lower because the class contains a public constructor. Nevertheless it is documented by the developers to be an implementation of the Singleton pattern.

The fuzzy rating of the get method is 1.0 in both cases as the get method implementations contain only one statement each.

#	Rating	Roles	Documented in	Evaluation
		Singleton		
1	95,85%	Clipboard	TCSH06, Javadoc	True Positive
2	82,52%	Iconkit	TCSH06, Javadoc	True Positive

Table 8. Detection Results for the Singleton pattern

4.7 State / Strategy

State : "Allow an object to alter its behavior when its internal state changes. The object will appear to change its class." [7]

Strategy : "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [7]

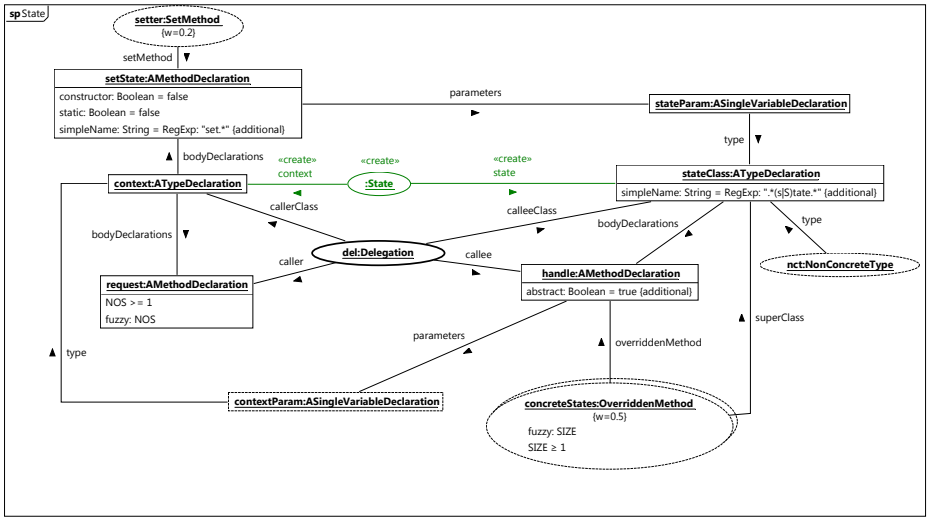

Fig. 10. Reclipse Specification of the State Pattern

Specification The State pattern is specified as follows. There is a stateClass and a context. The context has a request method which, whenever it is called, delegates to the stateClass's handle method. The context may pass itself as a parameter to the handle method as indicated by the additional contextParam object. The context also has a setState method to set the current state which takes an object of type stateClass as parameter. In addition, the stateClass and its handle method should be overridden more than once. This is specified constraint $SIZE \geq 1$ in the OverriddenMethod annotation set.

For the OverriddenMethod annotation we used the same fuzzy set rating function as for the Observer pattern (cf. Section 4.5, Figure 7).

As the Strategy pattern is structurally similar to the State pattern the two specifications are nearly identical. There is also a context class with a request method which delegates to the algorithmMethod of the strategyClass (instead of the handle method of the stateClass). The set method in that case is called setStrategy and takes a strategyClass as parameter. The algorithmMethod may take the context as a parameter and should be overridden more than once.

Detection Results Due to the structurally identical specifications, we could not distinguish State and Strategy pattern occurrences. To do this Reclipse supports a behavioral analysis which extends the structural analysis but also takes runtime behavior into account. [20] [21] [22] [23]

Table 9 shows a selection of our results for the State/Strategy pattern. The highest rated result is an actual Strategy pattern implementation as documented in the JHotDraw Javadoc. Pattern occurrence #3 is also a true positive. According to other publications, the role of the context in these two cases is played by the class DrawingView which is an interface implemented by the class StandardDrawingView.

Pattern occurrence #2 is in our opinion a true positive that is not documented as a pattern implementation. Its structure is almost identical to the structure of occurrence #1, hence the high rating. We manually inspected the code and found out that a PolyLineFigure has up to two LineDecorations that determine how a line's ends are to be drawn. When drawing the line's ends the drawing operation is delegated to the LineDecorations if available. Depending on which LineDecoration is set for a line's end the shape of the line end differs. Classes implementing the interface LineDecoration implement the draw operation and determine the shape. All these facts together qualify the pattern occurrence #2 to be a Strategy pattern implementation.

The remainder of the detected occurrences consists of false positives, albeit with a lower rating. It is noteworthy that all occurrences detected by Reclipse were also reported by Tsantalis et al.[7]

[7] http://java.uom.gr/∼nikos/pattern-detection.html

#	Rating	Roles		Documented in	Evaluation
		Context	State/Strategy		
1	74,32%	StandardDrawingView	Painter	TCSH06, Rie00, Javadoc	True Positive (Strategy)
2	71,82%	PolyLineFigure	LineDecoration	TCSH06	True Positive (Strategy)
3	69,45%	StandardDrawingView	PointConstrainer	TCSH06, Rie00, Javadoc	True Positive (Strategy)
4	66,91%	StandardDrawingView	DrawingEditor	TCSH06	False Positive
5	61,88%	StandardDrawingView	Drawing	TCSH06	False Positive
6-16	58 - 44%	...	...	TCSH06	False Positives

Table 9: Detection Results for the State pattern

4.8 Template Method

"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." [7]

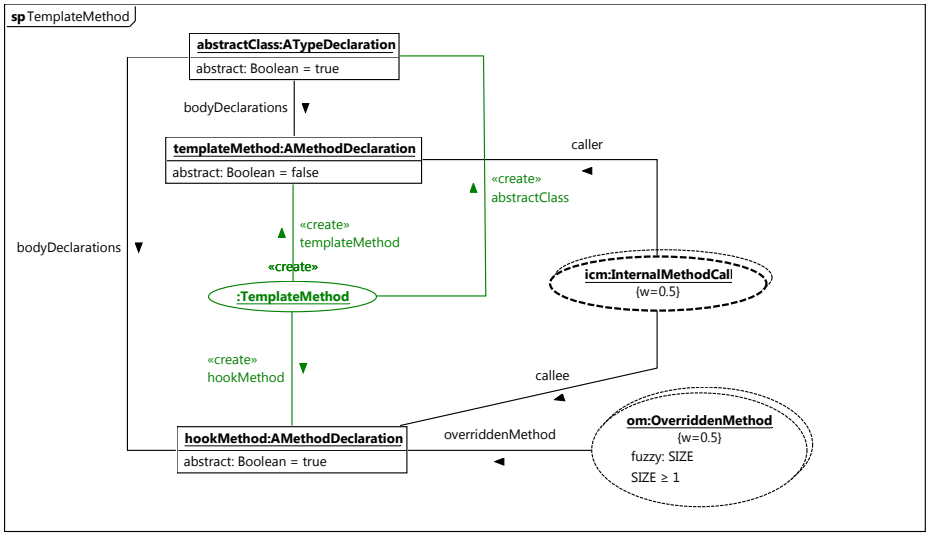

Fig. 11. Reclipse Specification of the Template Method Pattern

Specification Template Method is a relatively simple pattern. It contains an abstractClass with the actual templateMethod and several abstract hookMethods. The templateMethod should call the hookMethods as indicated by the InternalMethodCall. Those should also be overridden by subclasses of the abstract class.

For the OverriddenMethod annotation we used the same fuzzy set rating function as for the Observer pattern (cf. Section 4.5, Figure 7).

Detection Results Table 10 shows all 18 candidates we detected in a condensed form. Reclipse detected one occurrence of the Template Method pattern for each combination of the three roles Abstract Class, Template Method, and Hook Method. This is problematic, when one template method contains several hook methods (which is common for this pattern). Consider candidates 15 to 18. Reclipse detected four occurrences for the method paint in class PaletteButton. The hook methods paintBackground, paintNormal, paintPressed, and paintSelected are all called in paint, so this would be one occurrence of the Template Method pattern with four hook methods. A possible extension of Reclipse to handle these cases is discussed in Section 5.

Tsantalis et al. [19] reported all of our detected Template Method occurrences on their web site. If Reclipse's problems described in the last paragraph are considered, our detection results for this pattern match those of Tsantalis et al. completely.

#	Rating	Roles			Documented in	Evaluation
		Abstract Class	Template Method(s)	Hook Method		
1-4	82,84%	AbstractFigure	size, containsPoint, invalidate, center	displayBox	Javadoc, LDGR10, TCSH06	True Positive
5	82,05%	AbstractFigure	displayBox	basicDisplayBox	Javadoc, LDGR10, TCSH06	True Positive
6	81,12%	AbstractFigure	moveBy	basicMoveBy	Javadoc, LDGR10, TCSH06	True Positive
7	75,46%	AbstractHandle	displayBox	locate	TCSH06	True Positive
8-9	59,91%	ChangeConnectionHandle	source, invokeStart	target	TCSH06	True Positive
10-11	59,91%	ChangeConnectionHandle	invokeEnd, invokeStep	setPoint	TCSH06	True Positive
12	59,91%	ChangeConnectionHandle	invokeStart	disconnect	TCSH06	True Positive
13	59,91%	ChangeConnectionHandle	invokeEnd	connect	TCSH06	True Positive
14	56,65%	ActionTool	mouseDown	action	TCSH06	True Positive
15-18	56,65%	PaletteButton	paint	paintBackground, paintNormal, paintPressed, paintSelected	TCSH06	True Positive

Table 10: Detection Results for the Template Method pattern

5 Lessons Learned

The evaluation of our detection results showed that our specification language lacks a possibility to specify sets of objects. This is becomes apparent for example for the Command or the Template Method patterns. The problem exists for all pattern specifications which contain an abstract level that can be concretized multiple times, e.g., a template method which contains multiple hook methods. In this case, Reclipse will detect one pattern occurrence for each combination of elements, i.e. instead of reporting just one Template Method occurrence with four hook methods it will report four occurrences with one hook method each (cf. pattern occurrences #15 to #18 in Table 10).

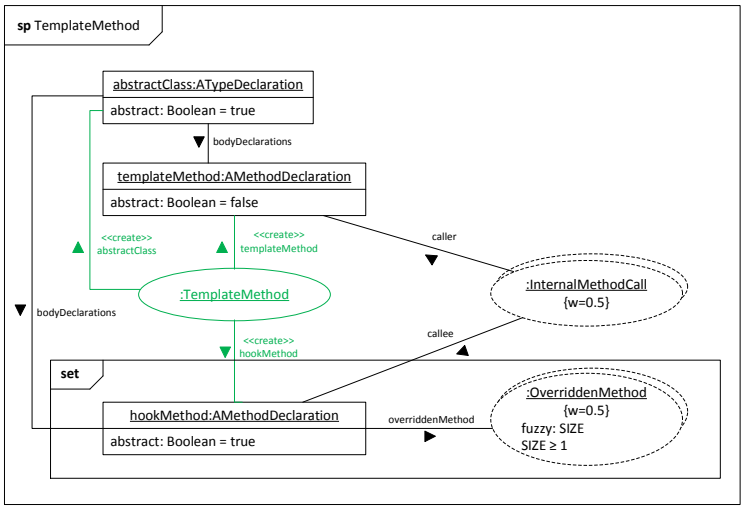

Fig. 12. The Template Method specification with a Set Fragment

This could be remedied by the introduction a special language construct which expresses that a certain subgraph of the pattern specification can occur multiple times. In the style of the already existing additional fragments, this *Set Fragment* is used in Figure 12 to express that a template method can have more than one hook method.

Another topic is the improvement of our pattern specifications. Specifying patterns is an iterative process. The quality of a given specification can only be assessed by looking at the detection results and going back to change details, add or remove constraints and manipulate weights. During the evaluation that is presented in this report, we continuously adapted our specifications. Still, changing some of the weights could further improve the ratings and separate true and false positives more clearly.

Furthermore, it should be assessed if and how this "fine-tuning" is affected by the analyzed system, i.e., are the pattern specifications in that exact form applicable to other systems or is the fine-tuning system specific.

6 Conclusions and Future Work

In this report, we presented an evaluation of a static pattern detection approach which uses additional specification elements, metrics and fuzzy expressions to rate the detection result. The rating reflects a detected pattern occurrence's adherence to the pattern specification and helps distinguishing true from false positives.

We applied our approach to the JHotDraw framework and compared the detection results to those of other approaches. It shows that we were able to detect a lot of true positives which were also described in other publications. In addition, the true positives for the most part achieved a higher rating than the false positives. Thus our approach provides a useful guidance for the reverse engineer in assessing the results of a static pattern detection. Instead of leaving him with just a set of equal results, we rank the detected pattern occurrences and thereby propose which occurrences are most like to be true positives.

It was already mentioned in the report that Reclipse also comes with a dynamic analysis to judge the static detection results based on the behavior [20] [21] [22] [23]. This potential could be applied to the results in this report in order to see which of the false positives can be ruled out that way.

Acknowledgments

We would like to thank Marie Christin Platenius for her help in carrying out our evaluation.

References

1. D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns – Best Practices and Design Strategies*. Core Design Series. Prentice Hall, Sun Microsystems Press, 2 edition, 2003.
2. F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture – A Pattern Language for Distributed Computing*, volume 4 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2007.
3. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture – A System of Patterns*, volume 1 of *Software Design Patterns*. John Wiley and Sons, Ltd, 1996.
4. E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.
5. J. O. Coplien and D. C. Schmidt, editors. *Pattern Languages of Program Design*, volume 1 of *Software Patterns Series*. Addison-Wesley, 1995.

6. J. Dong, Y. Sun, and Y. Zhao. Design Pattern Detection by Template Matching. In *Proceedings of the* 23^{rd} *Annual ACM Symposium on Applied Computing (SAC'08)*, pages 765–769, Fortaleza, Brazil, 2008. ACM Press.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
8. Y.-G. Guéhéneuc and G. Antoniol. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.
9. Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting Design Patterns. In *Proceedings of the* 11^{th} *Working Conference on Reverse Engineering (WCRE'04)*, pages 172–181, Delft, The Netherlands, 2004. IEEE Computer Society.
10. R. S. Hanmer. *Patterns for Fault Tolerant Software*. Software Design Patterns. John Wiley and Sons, Ltd, 2007.
11. M. Kircher and J. Prashant. *Pattern-Oriented Software Architecture – Patterns for Resource Management*, volume 3 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2004.
12. A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi. Improving behavioral design pattern detection through model checking. In *Proceedings of the* 14^th *European Conference on Software Maintenance and Reengineering (CSMR '10)*, Madrid, Spain, 2010.
13. M. Meyer. *Musterbasiertes Re-Engineering von Softwaresystemen (Pattern-based Reengineering of Software Systems)*. PhD thesis, University of Paderborn, 2009. In German.
14. J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards Pattern-Based Design Recovery. In *Proceedings of the* 24^{th} *International Conference on Software Engineering (ICSE'02)*, pages 338–348, Orlando, USA, 2002. ACM Press.
15. D. Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zrich, 2000.
16. D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2000.
17. N. Shi and R. A. Olsson. Reverse Engineering of Design Patterns from Java Source Code. In *Proceedings of the* 21^{st} *International Conference on Automated Software Engineering (ASE'06)*, pages 123–134, Tokyo, Japan, 2006. IEEE Computer Society.
18. I. Stürmer and D. Travkin. Automated Transformation of MATLAB Simulink and Stateflow Models. In *Pre-Proc. of the* 4^{th} *Workshop on Object-oriented Modeling of Embedded Real-Time Systems*, volume tr-ri-07-286, pages 57–62. University of Paderborn, 2007.
19. N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis. Design Pattern Detection Using Similarity Scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.
20. M. von Detten, M. Meyer, and D. Travkin. Reclipse - A Reverse Engineering Tool Suite. Technical report, Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, 2010.
21. 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*, 2010.
22. M. von Detten and M. C. Platenius. Improving Dynamic Design Pattern Detection in Reclipse with Set Objects. In *Proceedings of the* 7^{th} *International Fujaba Days*, pages 15–19, Eindhoven, The Netherlands, 2009.

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

A Subpattern Specifications

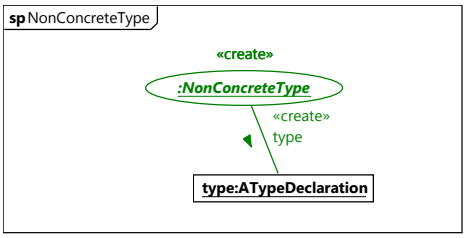

Fig. 13. The NonConcreteType subpattern (abstract)

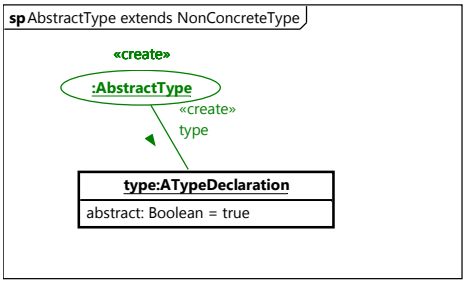

Fig. 14. The AbstractType subpattern (extends NonConcreteType)

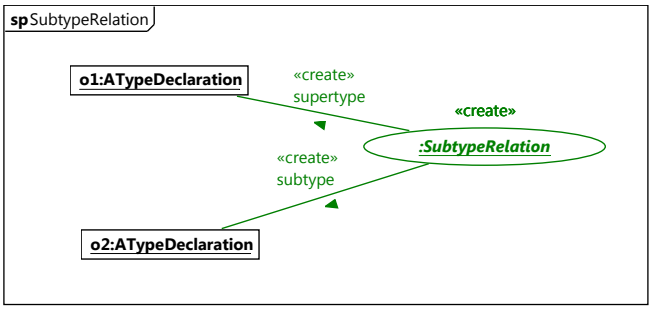

Fig. 15. The SubtypeRelation subpattern (abstract)

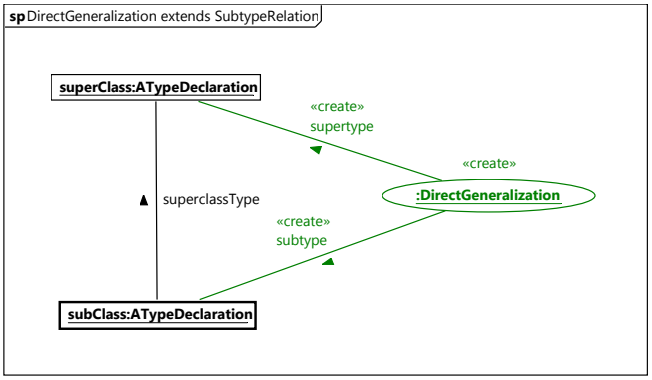

Fig. 16. The DirectGeneralization subpattern (extends SubtypeRelation)

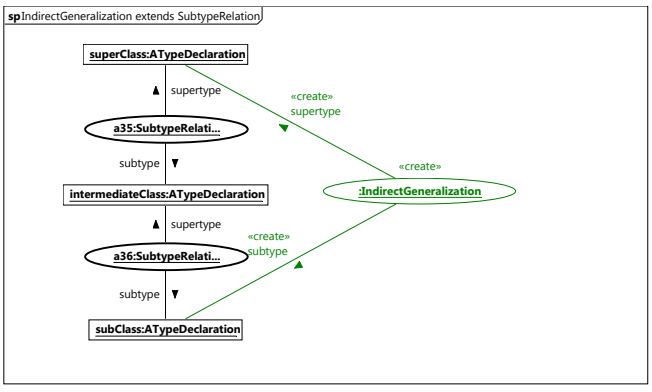

Fig. 17. The IndirectGeneralization subpattern (extends SubtypeRelation)

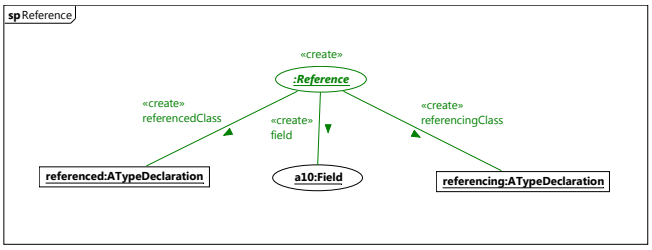

Fig. 18. The Reference subpattern (abstract)

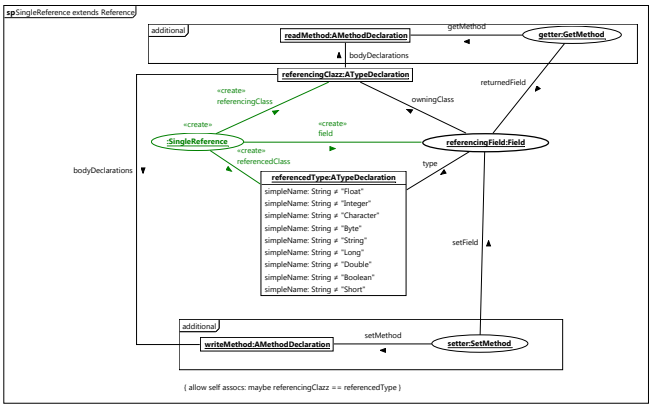

Fig. 19. The SingleReference subpattern (extends Reference)

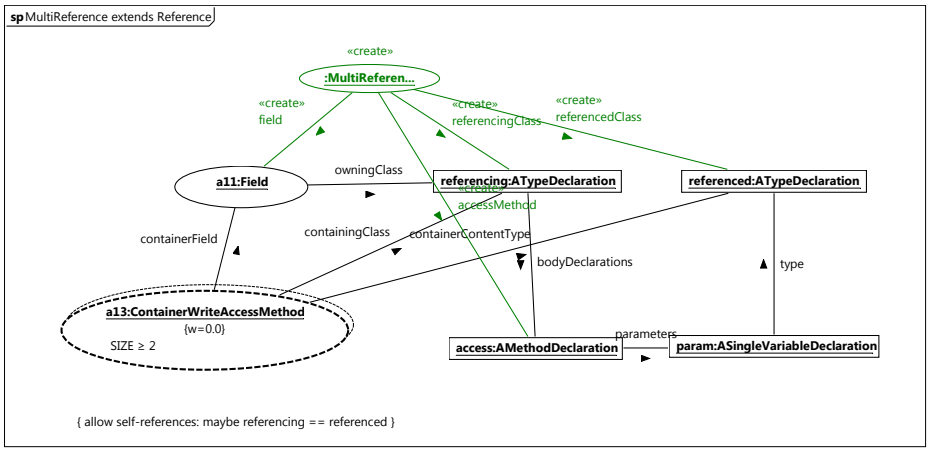

Fig. 20. The MultiReference subpattern (extends Reference)

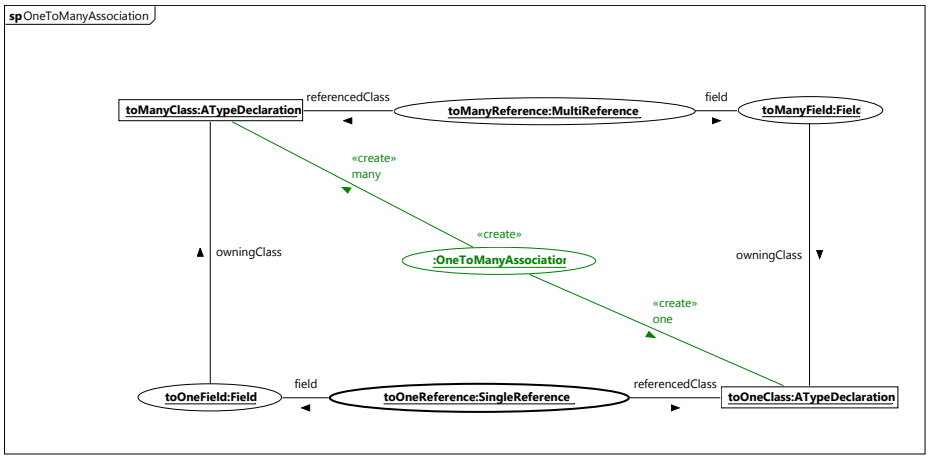

Fig. 21. The OneToManyAssociation subpattern

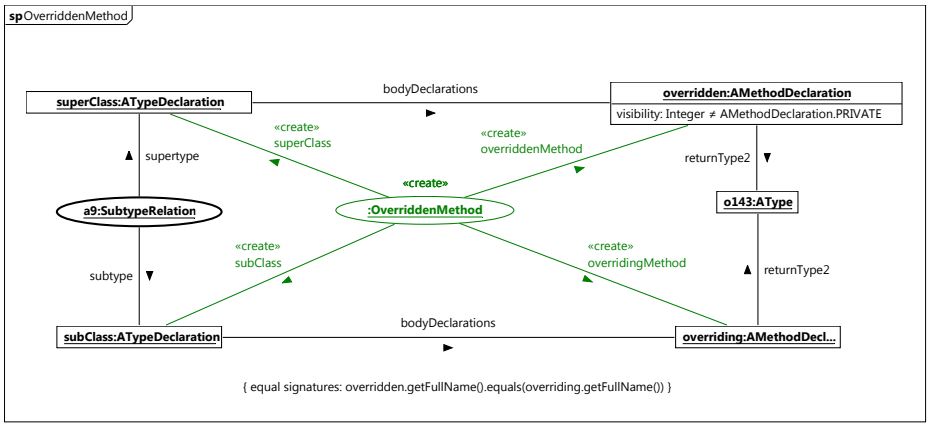

Fig. 22. The OverriddenMethod subpattern

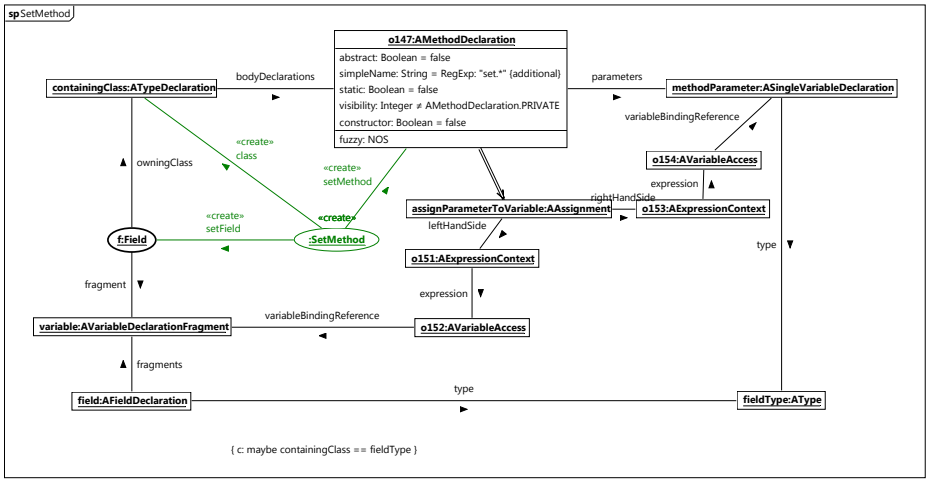

Fig. 23. The SetMethod subpattern

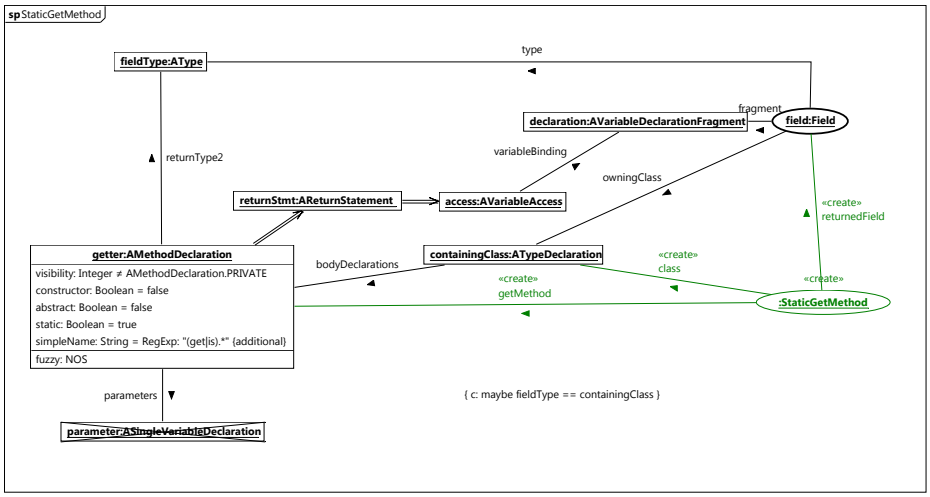

Fig. 24. The StaticGetMethod subpattern

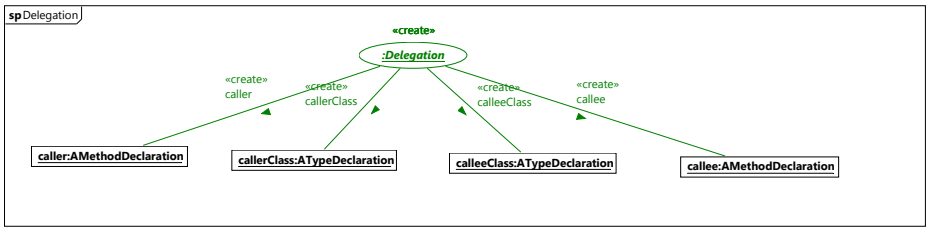

Fig. 25. The Delegation subpattern (abstract)

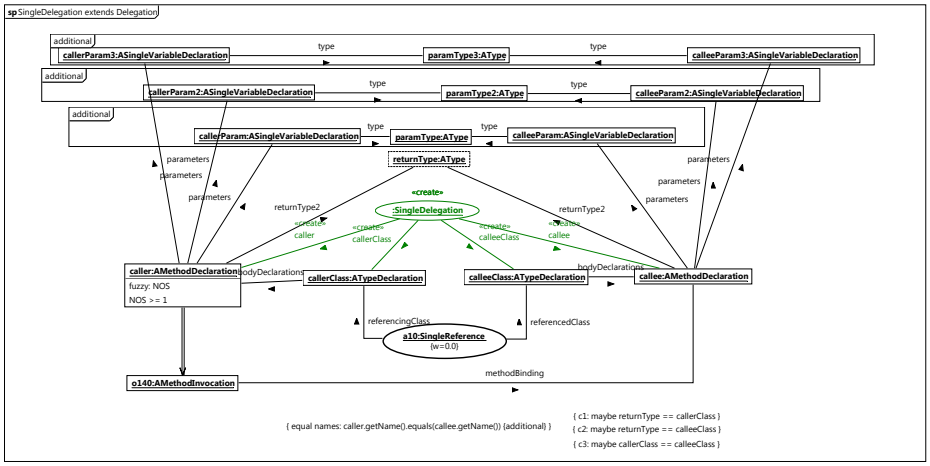

Fig. 26. The SingleDelegation subpattern (extends Delegation)

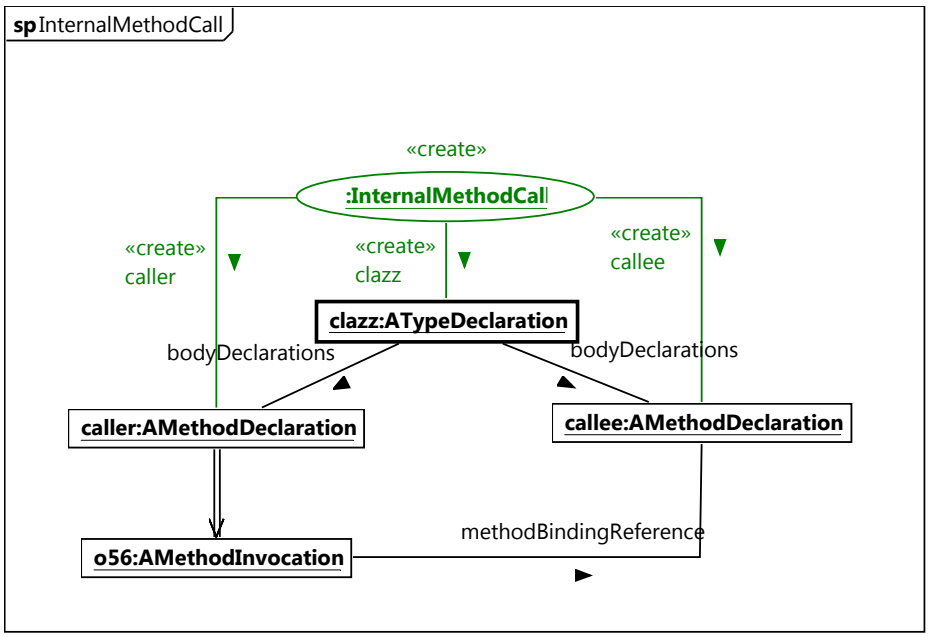

Fig. 27. The InternalMethodCall subpattern

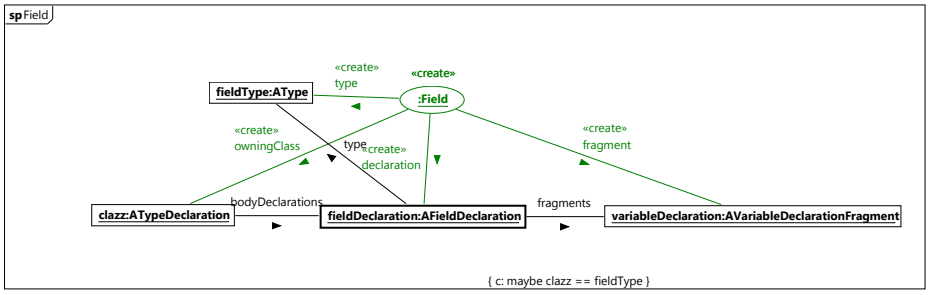

Fig. 28. The Field subpattern

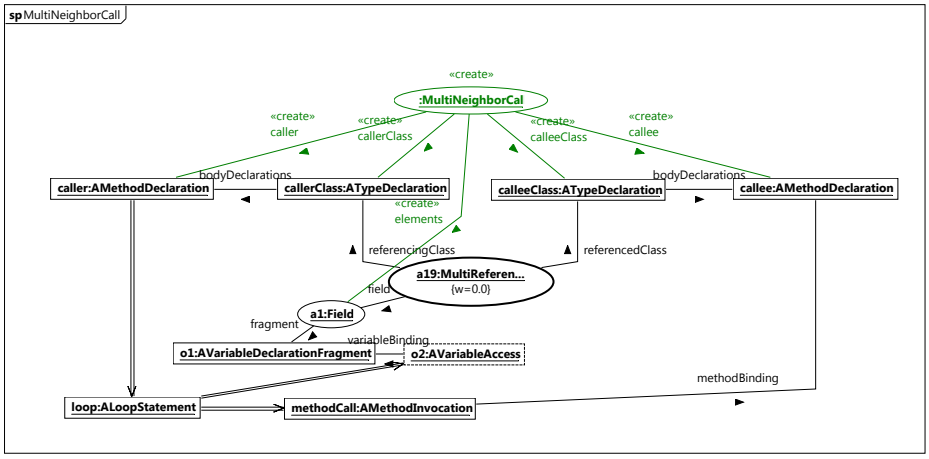

Fig. 29. The MultiNeighborCall subpattern