Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection

Article · May 2009

CITATIONS
10

READS
499

2 authors:

Alexander Binun
Ben-Gurion University of the Negev
9 PUBLICATIONS **92** CITATIONS
SEE PROFILE

Günter Kniesel-Wünsche
University of Bonn
92 PUBLICATIONS **1,442** CITATIONS
SEE PROFILE

Institut für Informatik III

Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection

Alexander Binun, Günter Kniesel

Technical Report IAI-TR-2009-01, January 2009

Updated April 2009

Forschungsbericht

Technical Report

ISSN 0944-8535

Forshungsberichte sind erhältlich durch:
Sekretariat für Forschungsberichte
Institut für Informatik
Universität Bonn
Römerstraße 164
D-53117 Bonn
Fax: (0228) 73-4382
E-mail: techrep@cs.uni-bonn.de

Technical reports can be requested from:
Technical Reports Secretary
Institute of Computer Science III
University of Bonn
Römerstraße 164
D-53117 Bonn
Fax: (0228) 73-4382
E-mail: techrep@cs.uni-bonn.de

Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection

Günter Kniesel, Alexander Binun
{gk,binun}@iai.uni-bonn.de
University of Bonn
Institute for Computer Science III
Römerstr. 164, D-53117 Bonn, Germany

Abstract

Identification of design patterns can deliver important information to designers. Therefore, automated design pattern detection (DPD) is highly desirable when it comes to understanding unknown code. In this paper, we present the results of evaluating five existing DPD tools (Fujaba, PINOT, Ptidej, DP-Miner and SSA) on various Java projects. These tools jointly cover almost all known DPD techniques but mix them in different ways, focusing on different indicators for the supported design patterns. Given that each tool designer is forced to make a choice of available techniques, trading precision for recall or speed (or vice-versa), we argue that instead of updating a concrete DPD tool it would be preferrable to fuse the outputs of several different DPD tools.

These results motivate our proposal of a novel DPD approach based on data fusion. It combines design pattern candidate sets coming from different tools implementing different DPD techniques. We show that a design pattern canwitnessfor the existence of another pattern. The particular strength of the witness-based approach is its ability (1) to provide correct diagnostics even if the inputs from the evaluated tools were partly wrong and (2) to detect even patterns that the individual tools do not identify or do not support at all. For the Decorator, Visitor and Observer pattern, the witness-based approach yields better precision and recall than provided by any single tool. In particular, it detects 24 out of 30 instances of these patterns missed in our experiments by the evaluated tools. These results motivate our suggestions for a set of improvements of existing DPD tools that will ease automated data fusion in the future. We also show that on the analysed instances of the Bridge, Mediator and Facade pattern, data fusion could not improve results, demonstrating that research into improvements of the basic detection techniques is still necessary.

1 Introduction

Design patterns [10] are domain-independent descriptions of groups of communicating classes that present solutions to recurrent object-oriented design problems. Uncovering design patterns from source code is therefore important for program comprehension.

Design pattern detection (DPD), like any information retrieval task, suffers from *false negatives* (missing correct instances) and *false positives* (incorrect classification of pattern instances) - see [7]. When comparing tools on the same input project it is commonly said that a tool that yields less false negatives has better *recall* and one that yields less false positives has better *precision*. These are conflicting issues. DPD tools that apply liberal detection criteria increase recall. On the flip side, they decrease precision. In contrast, tools that apply too restrictive detection criteria filter out many false positives, but correct instances may be filtered out as well. In addition, the need for speed adversely affects precision as well as recall.

Striving for a compromise between precision, recall, speed, or just for the sake of simplicity, existing DPD tools deliberately do not implement all existing pattern detection techniques or do not implement them to their full extent. Therefore, they often render different, contradictory results on the same code, as shown by the state of art overview of Dong and Peng [7] and by our own practical evaluation of five existing tools (Section 3).

These observations motivate the basic hypothesis underlying our approach: Instead of striving for the (impossible) ideal DPD tool it is more promising to combine the expertise of existing tools implementing different DPD approaches. Each tool, analyzing certain aspects of a pattern, contributes its estimates of which program elements are likely to form a particular pattern instance. We call this the *data fusion approach to DPD*. It builds on the synergy of proven techniques without requiring any expensive reimplementation of what is already available and has proven effective. In addition, it can benefit from future advances in the detection quality of any of the basic tools, no matter by which techniques they are achieved. It should be noted that the data fusion approach in text search engines was advocated already in 1980-s (see , for example, Croft [4]).

In order to verify our claim, we evaluated several pattern detection tools and developed the *witness-based approach* for combining their results. The essence of witnesses is that a diagnosed design pattern can be an indicator for the existence of another pattern. Witnesses leverage previous work on elemental design patterns [26] and the subpattern relationship (Zimmer [32] calls them "used-by" relationship) from the level of very fine-grained structures, to the level of full-fledged design patterns. The particular strength of the witness-based approach is its ability (1) to provide correct diagnostics even if the inputs from the evaluated tools were partly wrong and (2) to detect even patterns that the individual tools do not support. For the Decorator, Visitor and Observer pattern, the witness-based approach yields better precision and recall than provided by any single tool. In particular, it detects 24 out of 30 instances of these patterns missed by the evaluated tools. Our contributions include

- a practical evaluation of existing DPD tools for Java,
- a novel approach to design pattern detection based on data fusion, the notion of witnesses and criteria for the combination of witnesses.
- an evaluation of the strengths and limits of the witness-based approach to design pattern detection,
- suggestions for improvements of current DPD tools that will ease automated fusion of DPD results in the future.

The next section (Section 2) presents some basic notions and background work. Section 3 describes our practical evaluation of five existing tools and the experimental findings that motivated the fusion-based approach. Section 4 describes our approach to wittness-based design pattern detection. Section 5 presents the manual evaluation results obtained from the data fusion approach and from each individual tool. Section 6 discusses related work. Section 7 summarizes our contributions and outlines several future research directions. The Appendix section presents the true positives collected by us and false positives reported by different tools.

2 Background

This section reviews previous work on design pattern detection, extends basic definitions from literature that will be used later, summarizes related work and discusses the diversity of the results returned by different pattern recognition tools.

2.1 Basic Definitions

Patterns and roles A *design pattern* describes alternative solutions to recurring design problems. The description of a solution comprises a set of *roles*. Roles are duties that can be fulfilled by program elements (types, methods, fields), their relations (inheritance, subtyping, aggregation, association) or their collaborations (expressed by

fragments of code or of dynamic UML diagrams). For example, in the Decorator pattern (Figure 8), the type known to clients plays the Component role. A class providing extra functionality and forwarding requests for Component operations to a Component instance plays the Decorator role. Any forwarding method plays the Operation role. The association referencing the object to which Operation invocations are forwarded plays the component role. Even the tiny code fragment of the forwarding call plays an essential role - without it, the pattern would not fulfil an important part of its purpose. This is an example of an unnamed role.

Instances and Candidates An *instance* of a design pattern *P* is a set of program elements that together play some roles of *P*. A pattern *candidate* is a set of program elemens that might be an instance.

Deviations and Overlapping Original pattern descriptions are rather informal, so programmers need to adapt them to specific situations [10]. The result is a high number of design pattern implementations that do not follow strictly the implementation variants discussed in the literature. We call them *pattern deviations*[1] to distinguish them from *pattern variations* [17, 28, 7], which include any variations, not just the non-standard ones. For example, a class playing the Leaf role in the Decorator pattern does not necessarily need to be a concrete class but can also be an abstract class or an interface (we will see this in Section 4). This was not discussed in the literature but nevertheless occurs in the benchmarks observed by us so we assume this is deviation. But in the Observer pattern the Observer class frequently receives the information about the state change from the Subject class and does not need to ask it for the additional information (this is referred to as Pull Observer in [10]) so we consider Pull Observer as a variation. Sometimes in the Observer pattern Subject has only one Observer (for example, org.jhotdraw.util.PaletteListener as Observer, org.jhotdraw.util.PaletteButton as Subject) , we considered this as deviation (and refer to as *1:1 Observer*).

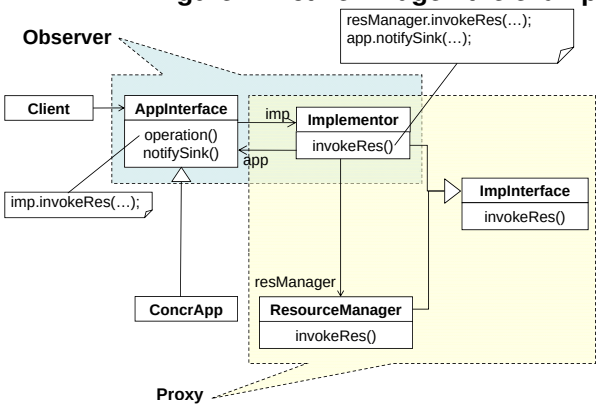
Pull Observer is an example of a *property relaxation*. Another typical deviation is *indirection* through additional program elements not described in the pattern. Indirection is possible for all transitive relationships (subtyping, inheritance, method calls or field accesses). Data flow through a series of field accesses, variables or method calls is a hard to detect, generic example of indirection. Smith et al. [26] and Kaczor et al. [13] provide examples of indirection. A more concrete one is the *Publish-Subscribe Observer* [10], which is a deviated Observer where subscription and notification functionality is placed in an additional SubscriptionManager between the subject and observer. The opposite of indirection is *role merging*, which occurs if the same program element plays different roles in the same pattern. This is different from *overlapping* [15], which occurs when a program element plays roles in different design pattern instances. Figure 1 illustrates the overlap of Observer, Proxy and Bridge of the Active Bridge [22] pattern. The class Implementor plays the Subject role in the Observer pattern (notifying the application of any access to resources). At the same time it plays the Proxy role in the Proxy pattern. The additional functionality of the Proxy instance is to notify the application of any resource accesses.

Property relaxation, indirection, role merging and the lack of some optional roles are the basic deviations that we regard in this paper. We claim that most deviations can be expressed as combinations of these basic deviations. So far, all our experiments confirmed this claim. Deviations and overlapping are the main challenge for DPD approaches. What is to be considered a typical instance of a pattern, a deviated instance or no instance is hard to capture as generally applicable pattern detection criteria [26]. Any criterion is bound to be either too general, accepting wrong results, or too strict, excluding correct ones.

Confidence Tools usually rank their results, expressing their confidence that a particular candidate is actually a correct instance by a score. Scores are typically expressed by a percentage, with 100% indicating total confidence.

[1] Guéhéneuc et al. [11] call them *distorted micro-architectures*.

Figure 1. Active Bridge - the example when Proxy, Bridge and Observer overlap



		What to Analyze	
		Structure	Behaviour
How to analyze	Static Analysis (program)	types fields and method signatures associations, inheritance	call graph data flow
	Dynamic Analysis (traces)	object relationships	execution traces object creation

Table 1. Basic DPD techniques

2.2 Basic DPD Techniques

Most design pattern detection algorithms match descriptions of the structure and the behaviour of sought patterns to information about the structure and behaviour of elements in a given program. Basic DPD techniques can be categorized in two orthogonal dimensions, with two subcategories each:

- **What to analyze**. *Structural analyses* deal with the structure of types (method signatures and variables) and inter-type relationships (associations, subtyping and inheritance). *Behavioral analyses* deal with program behaviour (for example, data and control flow, object creation).
- **How to analyze**. *Static analyses* analyse a program without executing it. *Dynamic analyses* are based on performing several trial program runs and collecting information from the execution traces.

Table 1 illustrates the four possible families of *basic DPD techniques* (from static structural to dynamic behavioural). Each table cell shows the information that is gathered by the respective technique.

2.3 Improving Precision

Back in 1998 Antoniol, Fiutem and Cristoforetti [2] noted that false positives were delivered by pattern detection tools because of insufficient behavioral analyses. Compared to early approaches, which mainly applied static structure analyses, precision is improved in modern approaches by additionally applying at least one of the following techniques.

- *Static control- and data flow analyses* [23] help to approximate program behavior more reliably thus eliminating false positives. This is an effective but potentially expensive approach.

- *Dynamic analyses* [31, 29] improve precision by matching selected execution traces against sequence diagrams of design patterns. This is also effective, in particular when applied in addition to structural criteria. Its limitation lies in the impossibility to cover all execution traces, possibly missing relvant behaviour.

- *Metrics-based fingerprinting* [12, 2] is based on the observation that for a concrete pattern, collaboration between participating classes is reflected by some OO metrics (cohesion, coupling) of these classes. Classes whose metric value is untypical for a certain pattern are discarded. This approach, however, works only for small programs where pattern instances do not overlap. During maintenance, many unordered entities and relationships are added. This leads to a lot of overlapping pattern instances [15] so that metrics-based fingerprinting becomes ineffective.

All the techniques for improving precision essentially strengthen the constraints that must be fullfilled for accepting a particular candidate as a true pattern instance. Unfortunately, this tends to miss deviated instances, reducing recall.

2.4 Improving Recall

Current techniques to improve recall in spite of pattern deviations include:

1. **Decomposition of patterns** into *elemental design patterns* (*EDPs)* [26] such as RedirectInFamily (see [24]). EDPs are so small that there are hardly any opportunities for deviations. Therefore they are easily and unambiguously recognized by the basic DPD techniques. EDPs are used in SPQR [26], Fujaba [17], EDPDetector4Java[3] and others.

2. **Using similarity** between the expected and the found structure and behaviour as a criterion (instead of precise matching). For instance, the Similarity Scoring Approach (referred to by us as SSA[28]) measures similarity between the graph representation of the sought pattern and of the analysed program by computing their graph product. Columbus [8] applies machine learning and measures similarity between the metric values of the classes got from training and the real values.

3. **Supporting some of the basic deviations** defined in Section 2.1. For instance, SPQR [26] supports indirection and Ptidej [11] supports property relaxation. We did not find any tool that supports role merging generically but only in special cases[2].

4. **Relaxed role checking** allows weaker constraints to be imposed on some roles or even allows some roles to be missing in pattern instances. For example, SSA [28] detects several Decorator instances in Java AWT although these instances do not include program elements playing the roles Concrete Decorator or Concrete Component. This is similar to our notion of optional roles.

5. **Relaxed delegation checking**. Some tools (e.g. Ptidej [11]) are restrictive when they require delegation calls to be made only through association links. The true cases when the pointer to the delegatee is returned by a getter method are missed. To accept such misses, DPMiner [6] requires the delegator code to include only the invocation of the delegatee method (with proper parameter type checking). The SSA tool [28] reduces the delegation check to the association check between the caller and callee classes. Both simplifications increase recall at the expense of many false alarms.

6. **Constraint relaxation** is based on formulating pattern recognition as a constraint satisfaction problem, with the possibility to relax some constraints, i.e. replacing them by weaker ones (Ptidej, [11]). Ptidej, for example, allows Leaves in the Composite pattern to be interfaces instead of concrete classes. In such a

[2]For instance, Ptidej and Fujaba are able to identify Command instances where the roles Command and Receiver are merged.

way, some deviated Composite instances can be recognized, as we will see in Section 4. The paragraphs 3 to 5 in this enumeration can be considered special instances of constraint relaxation. However, relaxation is not always the best approach. For instance, relaxed delegation checking would better be replaced by a sufficiently strong data flow analysis, thus capturing deviated cases but avoiding false positives. Semantically, this still is a relaxation but one that is achieved technically not by ignoring some stated constraints, but by replacing them by more appropriate ones. Our conclusion is confirmed by Dong and Peng [7] who note that existing pattern recognition tools suffer from wrong results mainly because of insufficient data flow analysis.

2.5 Improving Speed

Antoniol, Fiutem and Cristoforetti [2] note that checking all possible pattern candidates to detect a structural pattern of the cardinality k in the program consisting of n classes requires in the worst case $O(n^k)$ steps. The authors argue that the set of possible pattern candidates should be reduced by prior metrics-based filtering in order to decrease the detection complexity. The metrics-based fingerprinting approach of Gueheneuc et al. [12] also has the effect of speeding up the detection process by reducing the search space. Dong and Peng [6] and also Tonella et.al. [27] use variable and class names in conjunction with concept analysis in order to select the candidates which are more likely to be true positives. This is a good example that reducing the pattern candidate set can lead to better precision. If the reduction can be achieved by simple, efficient checks, it can also improve detection speed. However this is not always the case.

Wendehals [29] notes that polymorphism and dynamic binding aggravate the detection complexity and additionally lead to many false positives if static analyses are applied only. Wendehals suggests to use run-time analysis to reduce the number of pattern candidates and discard false positives stemming from polymorphism. However, correct pattern instances can be missed since dynamic analyses cannot cover all execution traces.

2.6 The DPD Dilemma

As we have seen so far, the different ways to achieve precision, recall and speed negatively influence each other, as illustrated in Figure 2. We call it the "Bermuda Triangle of design pattern detection", because it illustrates the inherent dilemma of DPD: The impossibility of achieving perfect results in more than one discipline. Any DPD tool that adventures alone on this sea in search of the holy grail of pattern detecton risks to sink in the storm of the many mutual interdependencies of its employed techniques. Put in a less metaphoric way, it is a matter of fact that, striving for a compromise between precision, recall, speed, or just for the sake of simplicity, existing DPD tools deliberately do not implement all existing pattern detection techniques or do not implement them to their full extent. Therefore, they often render different, contradictory results on the same code, as shown by the state of art overviews (see [7] and [9]) and by our own practical evaluation of five existing tools (Section 3). We conclude that there is currently no ideal pattern detection tool and there can never be one.

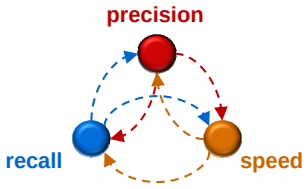


Figure 2. The Bermuda Triangle of design pattern detection: Improving any of the three parameters negatively affects the other two.

	Static structural	Static behavioral	Dynamic structural	Dynamic behavioral	Detected Deviations
SSA	Associations, inheritance, signatures, method calls, object creation	No	No	No	Up to 1 edge
DP-Miner	Associations, inheritance, signatures, names	Control flow and data flow	No	No	No
PINOT	Associations, inheritance, signatures	Control flow and data flow	No	No	No
PTIDEJ	Associations, inheritance, signatures, method calls, object creation	No	Association versus aggregation	Control flow and data flow (partial)	Attribute relaxation
FUJABA	Associations, inheritance, signatures	No	Association versus aggregation	State versus Strategy	Attribute relaxation

Table 2. Analysed tools and the basic techniques that they apply.

2.7 Reasons for Data-Fusion-Based DPD

Instead of striving for the impossible ideal DPD tool we suggest that it is more promising to combine the expertise of existing tools. Each tool, analyzing certain aspects of a pattern, contributes its estimates of which patterns are likely to be relevant. We call this the *data fusion approach to DPD*. It builds on the synergy of proven techniques without requiring any expensive reimplementation of what is already available. In addition, it can benefit from future advances in the detection quality of any of the basic tools, no matter by which techniques they are achieved. Last but not least, data fusion allows to experimentally evaluate combinations of techniques that any single tool cannot or could only after significant efforts for implementing techniques that other tools already incorporate. If the insights obtained this way with relatively small implementation overhead are valuable, they can still be included later in any individual tool.

In the next section we present the experimental findings that lead to our approach. Fusion-based DPD is introduced in section 4 and its effectiveness is evaluated in Section 5.

3 Tool Evaluation

Fusion-based DPD was motivated not just by the insights from the above review or related work but also by the results of a practical evaluation of five state of the art DPD tools. This evaluation also yielded the observations that showed us how to combine input from different tools. In this section we explain our selection of tools and benchmarks, give a short introduction to each tool and explain the findings that underly the fusion-based approach.

3.1 Evaluated Tools

As Poshyvanyk et. al. [21] note, "static analyses are conservative and sound while dynamic analyses are efficient and precise (given appropriate test suites). Thus, combining static and dynamic analyses compensate the imprecision of static analyses and the unsoundness of dynamic analyses." Accordingly, we tried to evaluate tools that incorporate as many basic pattern detection approaches as possible. In addition, we favoured tools that support pattern deviations.

From the 14 DPD approaches for Java listed in the state of the art overview of Dong et.al. [7], we found only 8 to be available for practical experimentation[3]. Based on the above criteria we selected the five tools listed in Table

[3]Three had only small, unavailable prototypes, two were licenced, one was implemented just for Linux.

	Classes	Lines of Code
Java IO library	108	11146
Lexi 0.1 (alpha)	127	6609
JUnit 3.7	157	4892
JHotDraw 5.1	173	8419
JHotDraw 6.0	406	21424
QuickUML 2001	224	9249
Java AWT	504	25754
PMD 1.8	520	4205
JRefactory 2.6	576	107200
JRefactory 2.8	1317	189028
Apache Tomcat 4.1.37	1371	309280
Eclipse platform 3.1	10680	799000

Table 3. The analyzed code repositories.

2. Together, they cover almost all basic pattern analysis techniques known currently[4]:

   - SSA (Similarity Scoring) [28] only performs static analyses of inter-class relationships and method signatures. SSA is able to tolerate small pattern deviations (up to one edge in the program graph).
   - DP-Miner [6] performs thorough static behavioral analyses. It does not support pattern deviations. No dynamic analyses are performed.
   - PINOT [23] performs thorough static behavioral analyses, which can sometimes be even a bit too restrictive. It does not support pattern deviations. No dynamic analyses are performed.
   - Ptidej [11] treats pattern recognition as a constraint satisfaction problem. Pattern deviations are modeled as constraint relaxation. For example, associations are accepted instead of aggregations but with lower score. Static behavioral analyses are limited to the identification of object creation. Dynamic control and data-flow analyses are supported to a certain extent.
   - Fujaba [17] decomposes patterns into EDPs (often called "subpatterns"), which enables recognition of rather deviated pattern instances, including cases of role merging and attribute relaxation. The most recent release of Fujaba, based on the PhD work of Wendehals [30, 31, 29], additionally analyzes selected program traces to filter out false positives.

DPD tools may assign scores to candidates, expressing their confidence that a particular candidate is actually a correct instance. Scores are typically expressed by a percentage, with 100% indicating total confidence.

3.2 Benchmarks

We performed an extensive evaluation of the above tools on several tens of true positive design pattern instances from the code repositories listed in Table 3. The analysed repositories cover a wide range of relevant scenarios. Java AWT, Java IO, Lexi, JUnit and JHotDraw are small libraries with mostly "clean" code, containing only a few deviations [5]. PMD and JRefactory are bigger frameworks which are used in the real scientific and industrial applications. Eclipse source code is an example of a large, quickly evolving industrial application containing very deviated pattern instances. This wide range was essential for realistic results since, according to our observations, tools that perform well on "clean" code can perform badly on deviated code[6].

[4] The sole exception are techniques currently supported only by the unavailable SPQR tool [26].

[5] JHotDraw, for instance, was accurately designed for teaching purposes (see the dissertation of Riehle, `www.riehle.org/diss`).

[6] It was also essential for evaluating speed and scalability. However, we do not address these issues here.

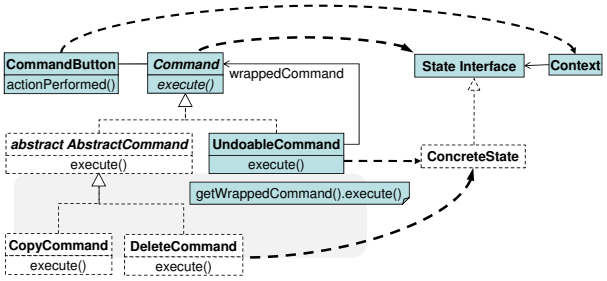


Figure 3. Decorator **instance classified as** State **by PTIDEJ and FUJABA**

3.3 Findings

First of all, our experiments confirm three main claims of Dong et.al. [7]:

- Different pattern detection tools render different results on the same code.
- No existing tool implements all known basic DPD approaches
- In particular, many failures are the result of insufficiently strong data flow analyses.

These observations motivated the investigation of data fusion (see also Section 2.6 and Section 2.7) . In addition, we made several observations, that pointed us to the criteria on which data fusion for DPD can be based. These are elaborated below.

3.3.1 A tool may recognize not the actual pattern, but a more general one

We observed that sometimes tools classify pattern instances as instances of more general patterns. For example, Ptidej and Fujaba often classify Decorator instances as State instances. Figure 6 illustrates such a case. It shows the classes of a Decorator instance from JHotDraw 6.0 and their inferred mapping to the roles of the State pattern. This common error of the two tools that use dynamic behaviour analysis results from the fact that Decorator resembles State not only structurally, but also behaviorally: Decorator mimicks State behavior since control traverses several nested Concrete Decorator objects and finally reaches a Concrete Component. This resembles the control flow in the State pattern when different states are switched. Obviously, the tools do not distinguish nested actions (the forwarding along a chain of decorators) from a sequence of actions (different invocations addressed in turn to different objects).

The Decorator instance in Figure 3 is deviated because the class playing the Leaf role is abstract. Our experiments yielded more misclassifications of the same type that occur for more or less deviated instances. We suggest that they are caused by the fact that only a subset of the constraints defining a pattern holds for deviated instances. If there is a pattern that is characterized by smaller set of constraints, it will be wrongly detected instead of the correct one.

3.3.2 Subpatterns and subpattern role mappings

In order to capture the relationship between the actual patterns and their wrongly recognized counterpart, we introduce the notion of *subpattern*: A pattern, B, is *a subpattern* of another pattern, A, if all instances of A are instances of B. In this case we also say A is a *superpattern* of B. For example, Decorator is a subpattern of State and Command is a subpattern of Visitor. Note that "superpattern" might seem counterintuitive, since it is usually a "smaller" pattern. However, this is consistent with the object-oriented terminology: superclasses are actually

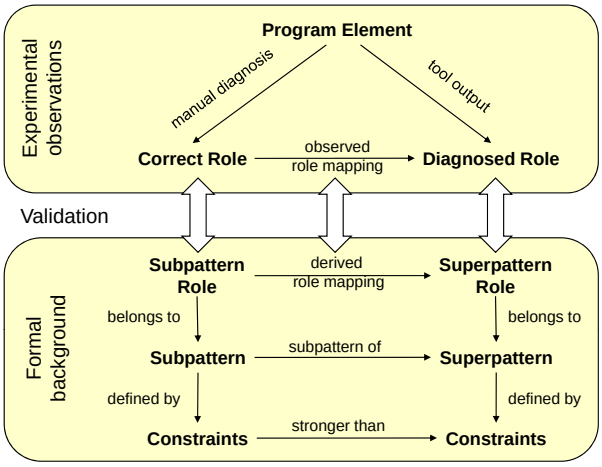


Figure 4. Validation of role mappings.

smaller classes (in terms of their contents) but their extent is bigger. This is also the case for superpatterns, which are often defined by less or weaker constraints but have a bigger extent too.

Note that the subpattern relation is purely technical and **does not imply that the patterns have the same intent**. For example, although Decorator is similar to State in that it switches control between different classes (Concrete Decorators and Concrete Component) the intention of Decorator is to allow run-time combination of additional functionality, not to enable dynamic behaviour change. Visitor could be seen as a more specific Command since Visitor may initiate very specific action for the kind of object it encounters - namely, to traverse the substructure of the encountered object.

The observation in Section 3.3.1 suggests that non-typical pattern variations are often recognized wrongly, as a superpattern instances. In order to verify this hypothesis, we derived the subpattern relation manually for all design patterns from [10] and compared the results to the ones obtained experimentally on the benchmarks described in Section 3.2. First, we specified roles and defining constraints (informally, according to the essence of each pattern desribed by Gamma et al [10]) for all the analyzed patterns. From these constraints we derived manualy the subpattern relations. Each subpattern relation implies a *role mapping*: The roles defined by some constraints in the subpattern are mapped to the roles defined by the same constraints (or a subset thereof) in the superpattern. This mapping is relevant for our empirical validation because the analyzed tools only report assignments of program elements to roles, not the constraints that justified these assignments[7]. Accordingly, we had to verify the consistency of each formally *derived role mappings* to the *observed role mappings*. The latter were obtained by running the tools and then mapping the manually determined correct roles to roles output by the tools. Figure 4 gives an overview of the described validation method.

Figure 6 illustrates the subpattern relation of Decorator to State, including the related role mappings. Figure 5 does the same for Decorator to Command. The respective role mappings are also presented textually in the first and second row of Table 4. The table presents all subpatterns relations for Decorator and Visitor along with the related role mappings and the constraints that are strengthened in the subpatterns. The first row of the table says that Decorator is a subpattern of Command, the Decorator role is mapped to Concrete Command, the Component role is mapped to Command and to Receiver, which must coincide, and the component field of the Decorator is mapped to the receiver field of the Concrete Command. The rows 4 to 7 show that subpattern role mappings may not be unique when different roles are defined by the same constraints. For example, in the Visitor pattern, the Visitor and the Element classes mutually call each other (double dispatch). Thus they can

[7] A notable exception is Ptidej. Its use of an explanation-based constraint solver lets it report constraints.

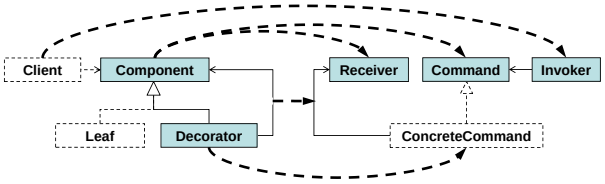


Figure 5. Decorator **is a subpattern of** Command

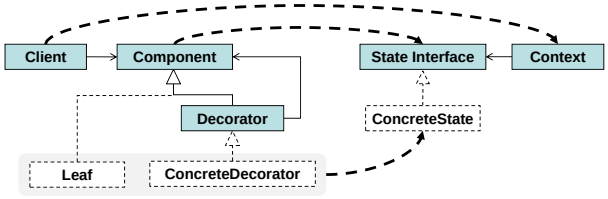


Figure 6. Decorator **is a subpattern of** State

both be interpreted as playing the role of Command (fourth and fifth row of Table 4).

Figure 7 illustrates all subpattern relations that we confirmed experimentally. It shows that the patterns Adapter, Command, State, Strategy and Template Method are more general versions of other patterns, although their intentions may differ. Smaller patterns are in the middle layer; the top and bottom layers are occupied by big patterns. It should be noted that some patterns (like Observer and Composite) have the same superpatterns (Strategy and Iterator). Only the role mappings differ. Technically, such patterns can be distinguished by analyzing EDPs like Inheritance or Association; see Section 4.6 for details. See also Section 4.7 about distinguishing the patterns within the same "witness group".

An arrow from pattern B to pattern A means that an instance of B has been classified as an instance of A (i.e. A is superpattern of B). If the arrow is solid, this finding is consistent with the conceptual classification of B as a subpattern of A. If the arrow is dashed, the classification is wrong, and does not correspond to the conceptual subpattern relation.

We distinguish between conceptual superpattern relations and superpattern relations created by concrete tool FPs. Strategy is a conceptual superpattern of Bridge, since the intent of Abstraction is switching between different Concrete Implementors (like Context switches between different Concrete Strategies). In contrast, Strategy is a "false positive" superpattern for Visitor. The reason is that Strategy is reported by SSA and Fujaba instead of Visitor (Concrete Element is Context, Visitor is Strategy interface). These tools report (erroneously) Strategy when ConcreteElements get a reference to Visitor interface and use polymorphism to choose an appropriate Concrete Visitor. The problem is that in the proper Strategy instance Strategy Context must be able to switch

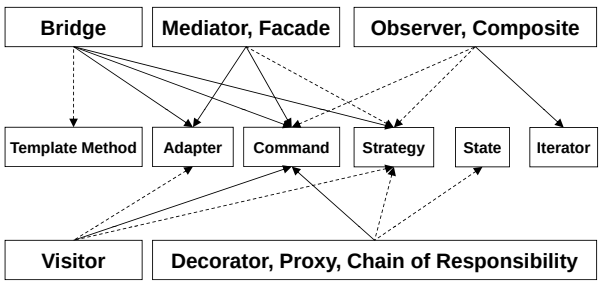


Figure 7. Subpattern relations: correct (solid arrows) and wrong classifications (dashed arrows).

Subpattern relation	Role mapping and constraints
Decorator → Command	**Decorator** → Concrete Command, **Component** → Command=Receiver **Decorator.component** → Concrete Command.Receiver
Decorator → State	Client → Context **Component** → State Interface Concrete Decorators → Concrete States
Visitor → State or Strategy	**Visitor** → Context **Element** → Interface Concrete Visitor, Concrete Element → Concrete State
Visitor → Command	**Visitor** → Command, **Element** → Invoker=Receiver
Visitor → Command	**Element** → Command, **Visitor** → Invoker=Receiver
Visitor → Adapter	**Element** → Client=Adaptee **Visitor** → Target Concrete Visitor → Adapter
Visitor → Adapter	**Visitor** → Client=Adaptee **Element** → Target Concrete Element → Adapter

Table 4. Subpatterns and role mappings for several big design patterns

between different Strategies. ConcreteElemens do not intend to switch between different Concrete Visitors. But even these "false positive superpatterns" are useful (for example, even a wrongly diagnosed Strategy indicates the presence of polymorphism by performing a particular action depending on the Visitor subclass).

Below is a brief description of conceptual superpattern relations for several big patterns:

- Bridge
  - Template Method is erroneously reported to be a superpattern of Bridge. It seems that the evaluated tools assume that Concrete Abstraction classes **override** methods defined at the top of Abstraction hierarchy that are further delegated to the Implementor hierarchy. However, Concrete Abstraction **only uses** these methods. But even wrongly reporting a Bridge as a Template Method is useful - this indicates the presence of an inheritance hierarchy.
  - Strategy is due to the potential of Abstraction (as Context) to switch between different Implementors; however it is hardly ever used. Obviously, Strategy diagnostics are mostly wrong for Bridge instances. An additional criterion could be that assigments to a field that are only carried out in constructors must not be interpreted as switching objects in the sense of changing strategies.
  - Adapter and Command show the fact that Abstraction forwards calls to Implementor.
- Mediator and Facade
  - Adapter reflects the fact that Facade adapts calls issued by a client to some Subsystem class. Though Facade may not change interface to a subsystem class (as the proper Adapter would do), we lean to consider Facade-Adapter as a true superpattern relation. In contrast, Mediator typically redirects calls from one Concrete Colleague to another one when all Concrete Colleagues implement the same Colleague interface (it may even broadcast a message to several Colleagues). Therefore we think that the observed Mediator-Facade superpattern relation is generated by false positives.
  - Command reflects that by holding the reference to Mediator (Facade), any colleague (Subsystem class) can use the functionality necessary to direct the call to a proper target (Receiver).
- Observer, Composite

- Iterator describes the iteration performed by Subject over Observers (in the Observer pattern) or by Composite (in the Composite pattern) over all children. To distinguish between the Observer and Composite patterns, one typically needs to check whether the Subject and Observer classes belong to the same hierarchy; if not, Observer is likely. In real code, Iterator is modeled by EDPs like Collection and Iteration (see [24]).
- Command and Strategy are sometimes reported as false positive superpatterns by Ptidej due to insufficient structural analysis. The reason is that Observer and Composite maintain a set of objects (not one object).

- Visitor

  - The fact that Command is a superpattern of Visitor stems from the observation that Visitor could be seen as a more specific Command since Visitor may initiate very specific action for the kind of object it encounters - namely, to traverse the substructure of the encountered object.
  - Adapter appears to be a superpattern of Visitor since the Visitor class adapts the accept() calls of some Concrete Element and redirects them to another Concrete Element. However, since both Concrete Elements typically implement the same interface, we lean to think that this observed superpattern relation stems from false positives. However, this false positive superpattern is useful since it indicates the presence of call redirection.

- Decorator, Chain of Responsibility, Proxy

  - Command demonstrates the fact that, by holding an instance of the Component class (which serves as Command), the Decorator class (Subject, Handler) performs the delegation.
  - State and Strategy. The identification of chains of Decorators and Chains of Responsibility as State reflect reflect false positives produced by tools based on dynamic behavioural analyses (Ptidej and Fujaba). They notice control flow passing from the action method of one instance to the next one in the chain, not distinguishing it from the switching of control that happens in a State scenario. They do not distinguish control passing *only* the objects in a chain (in the case of Decorator, Chain of Responsibility or Proxy) from control passing several objects including a Context instance (in the case of State or Strategy). This is a characteristic of switching State or Strategy instances because the method for doing it typically resides in the Context class. However, there are cases (in Eclipse core) when Concrete State may create itself a new state object and pass the control to it without notifying Context. Such deviations are also discussed by Odrowski et al (see [18]). But these cases seem to us to be rare and non-typical deviations.
    So we lean to think that Decorator-State subpattern relation is generated by false positive. Nevertheless this false positive is useful since it indicates passing control between different Concrete Decorators (Concrete Handlers etc).

3.3.3 Smaller Patterns are Identified More Reliably and with Higher Confidence

We say that a pattern is *smaller* if it is defined by a smaller number of constraints regarding program structure or behaviour. A smaller pattern does not necessarily need to be a superpattern. Unlike previously, we only consider the number of constraints, not whether they are in a subset relation. This may seem like a quite weak measure but it is sufficient for our purpose. We observed that

- the number of true positives among smaller patterns (Adapter, Command, State, Strategy, Iterator, etc.) is much higher than among bigger ones (Decorator, Chain of Responsibility, Mediator etc.).

- the analysed tools give smaller patterns higher scores than they assign to bigger patterns, usually 70-75% or higher;
- instances of smaller patterns are often classified identically by more than one tool (in around 80% cases).

Obviously, detection of smaller patterns is more reliable and the tools express their higher confidence in smaller patterns by the higher score. We suggest that the main reason behind both is that a smaller number of constraints must be satisfied. Furthermore, smaller design patterns are conceptually simpler and give rise to less deviations. This is consistent with the widely accepted EDP concept. Elemental design patterns are just *very* small patterns that are recognized with very high reliability and very high scores. They contain less behavioral aspects so they can be matched easily.

3.3.4 Big Patterns are Seldom Classified Identically

So far, we discussed that instances of big patterns are often *misclassified* as instances of superpatterns. A related finding is that instances of big behavioral patterns are rarely classified *identically* by several DPD tools. For instance, only 15% of the detected Visitor and 10% of the detected Bridge are classified identically. On the downside, identical classification is mainly confined to code that follows pedantically the implementation rules of the pattern description [10] - for example, the java.io package or JHotDraw, which was initially designed for teaching purposes. On the upside, however, instances of big patterns that are identically classified by more than one tool have a very high likelyhood of being true positives.

4 Data Fusion Approach

Striving for a compromise between precision, recall, speed, or just for the sake of simplicity, existing DPD tools deliberately do not implement all existing pattern detection techniques or do not implement them to their full extent. Therefore, they often render different, contradictory results on the same code, as shown in the previous section. Instead of striving for the ideal DPD tool we suggest that it is more promising to combine the expertise of existing tools. Each tool, analyzing certain aspects of a pattern, contributes its estimates of which patterns are likely to be relevant. We call this the *data fusion approach to DPD* since we fuse the outputs of several DPD tools. It builds on the synergy of proven techniques without requiring any expensive reimplementation of what is already available. In addition, it can benefit from future advances in the detection quality of any of the basic tools, no matter by which techniques they are achieved. Last but not least, data fusion allows to experimentally evaluate combinations of techniques that any single tool cannot incorporate (or could only after significant upgrading efforts). If the insights obtained by data fusion with relatively small implementation overhead prove valuable they can be included later in any individual tool.

Approach Overview The tool evaluation in the previous section showed that the results rendered by different tools implementing different detection approaches can be quite diverse. We think of a DPD tool as an expert examining the code from his own viewpoint. The insight that joining opinions of several experts is better than consulting only one opinion motivates the *data fusion approach* to DPD elaborated in this section.

We noted in Section 3 that different DPD tools often classify instances of Adapter, Command, State, Strategy, Iterator and Template Method identically, with high scores, high precision and identical role assignments for the most important roles. Bigger patterns (Observer, Visitor, etc.) can be also classified identically by several tools, but this occurs more rarely. Obviously, if different tools "agree" a true positive is more likely than when relying on the judgement of only one tool. This is the essence of *joint recognition* introduced in Section 4.3.

If several tools do not classify a pattern instance identically, we go to the next stage - checking whether the tools classify their *subpatterns* (Section 3.3.2) identically with consistent role assignments. Intuitively speaking, if tools

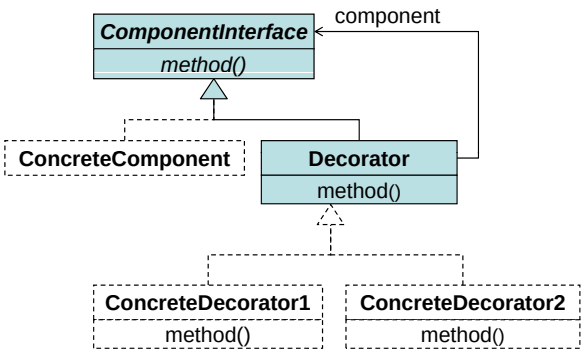


Figure 8. The Decorator **pattern. Mandatory roles are indicated by plain lines and dark background, optional ones by dashed lines and white background.**

do not agree on a complex pattern instance, we check whether they agree on different "witnesses" of the pattern. Then we are able to *reconstruct the pattern from witnesses*, as described in Section 4.4. Witnesses are introduced first, since they are the common basis of joint recognition and reconstruction.

Our approach is based on the following notions: witnesses, optional and mandatory roles, joint recognition and reconstruction of patterns from witnesses. These notions are introduced in this order in the remainder of this section.

4.1 Mandatory and Optional Roles

In this paper we distinguish between *mandatory and optional roles*. *Mandatory roles* represent the "essence" of a pattern. Each of the mandatory roles of a pattern *P* must be played by some program element in a candidate in order to consider it an instance of *P*. Roles that may be missing in an instance are called *optional*. From a pattern detection point of view they provide additional evidence that we indeed have an instance but their non-existence does not preclude identifying the instance.

For example, in the Decorator pattern all the above-mentioned roles are mandatory. They implement the basic concept of the pattern: performing some additional tasks on behalf of the Component class or forwarding calls to it. The roles Concrete Component and Concrete Decorator are optional because their implementation may be deferred. This is typical of frameworks, which only implement core functionality to be extended by clients.

4.2 Witnesses

A *witness* W of another pattern P is a pattern of which we know that some or all of its roles also play some role in P. Thus any instance of a wittness gives hints about the occurrence of an instance of the witnessed pattern. This is a rather general definition. In this paper we focus on a specific form of witnesses. Leveraging on the good recognition of small patterns (Section 3.3), we take superpatterns as witnesses of their respective subpatterns (Section 3.3.2). For example, Command and State are witnesses for Decorator because:

- **Command**. The Decorator class holds an instance of the Component class and invokes it (when the control is passed to the superclass of Decorator). This process is initiated by the invoker of a Decorator instance. This is also the behavior of the Command pattern. The invoker of Decorator plays the Command Invoker role, the Component class plays the Command Interface role (and at the same time the Receiver role), the Decorator class plays the Concrete Command role.

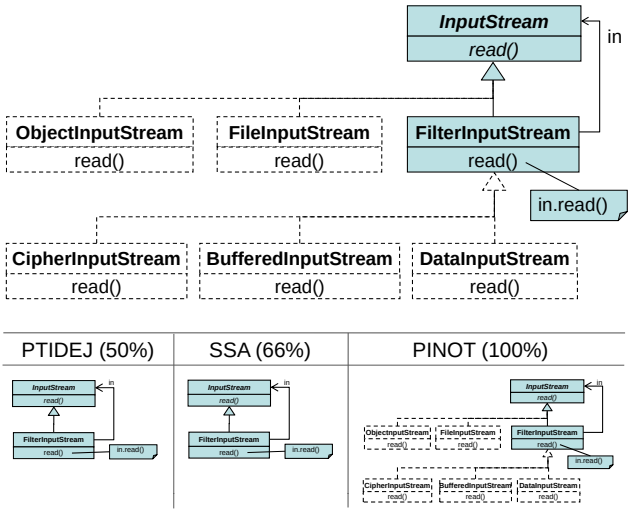


Figure 9. Three tools agree on this Decorator **instance**

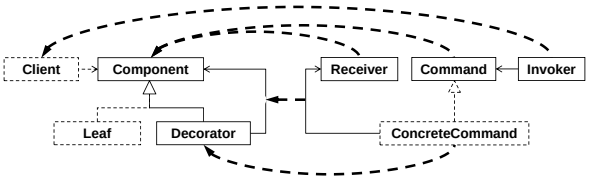


Figure 10. Command **is a witness of** Decorator

- **State**. When a Decorator instance is called, the control is passed (in turn) to several instances of Concrete Decorator (according to the order they were put into a "decorator chain"). Finally, a Leaf class gets the control. This resembles the behavior of State. which also lets control pass through different classes. The Component class plays the State Interface role, the invoker of Decorator plays the State Context role, Concrete Decorator and Leaf classes play the Concrete State role.

Figures 10 and 11 illustrate that, because superpatterns are taken as witnesses for subpatterns, the *witness role mappings* are inversed compared to the subpattern role mappings shown in Figures 7, 5 and 6. This has the implication that the witness role mapping might not be unique. Whereas in the case of Command the inversion still yields a unique mapping, the Concrete State role is mapped to a *set* of Decorator roles. In order to distinguish these we need to join the diagnostics of different witnesses as explained below.

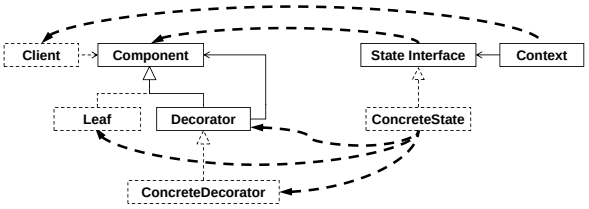


Figure 11. State **is a witness of** Decorator

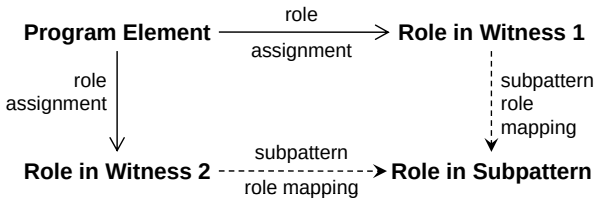


Figure 12. Consistent role mappings

4.3 Agreement and Joint Recognition

If two design pattern recognition tools, say *A* and *B*, classify the same set of program elements as an instance of the same pattern P with high scores (so far, we assume more than 50%) *and* their role assignments for the mandatory roles (see Section 2.1) are the same, we say that *A and B agree on this pattern instance*.

Figure 9 presents a Decorator instance from the java.io package on which PTIDEJ, SSA and PINOT agree with 50% score from Ptidej, 60% from SSA and 100% from PINOT[8]. Note that the agreement is not disturbed by the fact that only PINOT identifies the optional roles.

If at least two tools agree on an instance of a pattern P and other tools do not disagree on P's witnesses, we say that this pattern instance is *jointly recognized*. The Decorator instance from java.io (Figure 9) was jointly recognized by SSA, PINOT and PTIDEJ.

4.4 Disagreement and Reconstruction

Unfortunately, different tools agree on instances of *big* patterns quite rarely - mostly this occurs in well-designed libraries like $java.io$ or $JavaAWT$. However, we observed that in most cases tools disagree on big pattern instances but agree on most of their witnesses. For example, the evaluated tools disagree on more than 70% of all Decorator instances but agree on around 85% of Decorator witnesses. This observation motivates the following definition:

Reconstruction Let different tools agree with very high scores (so far, we assume at least 66%) on instances of patterns that witness pattern P. If the related role assignments and witness role mappings for all the witnessing instances are complete for P and consistent we say that P *is reconstructed from witnesses*.

Consistency Let I be an instance of the pattern W that witnesses pattern P . Let RA be the role assignment from program elements in I to roles of W and RM be the role mapping from roles of W to roles of P. Let $\overline{I}$ an instance for another witness $\overline{W}$ and $\overline{RA}$ and $\overline{RM}$ be the respective role assignments and role mappings. The different role assignments an role mappings are *consistent* if their composition maps each joint program element of I and $\overline{I}$ to the same role of P: $\forall e \in I \cap \overline{I} : RM(RA(e)) = \overline{RM}(\overline{RA}(e))$. This is illustrated in Figure 12.

Completeness Consider a set of different instances of witnesses with the related role assignments and role mappings defined as above. Let $CoveredRoles$ be the set of all roles of *P* to which any element of any instance is mapped by the composition of role assignments and role mappings. The role assignments and role mappings are *complete for P* if they cover all the mandatory roles of P, that is, if the mandatory roles are a subset of $CoveredRoles$.

[8]FUJABA does not recognize this Decorator and DP-Miner does not recognize decorators at all.

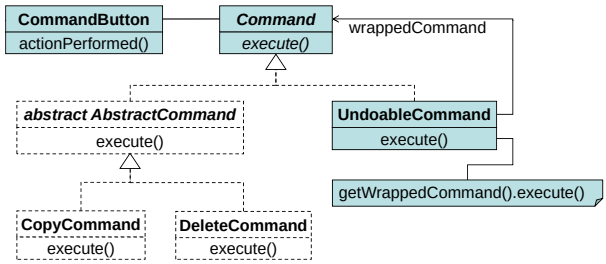


Figure 13. Tools disagree on this Decorator **instance**

Figure 13 presents a Decorator instance from JHotDraw 6.0 on which the tools disagree because it is deviated. However, this Decorator instance can be reconstructed from witnesses because three tools strongly agree on the existence of witnesses among the program elements in Figure 13.

PINOT and DP-Miner reject this instance because of the deviation - the Leaf role is played by an abstract class AbstractCommand instead of a concrete class. Fujaba and Ptidej reject this instance since they require the Decorator-Component delegation to be done directly through a field; they lack behavioral analyses to recognize the getter getWrappedCommand(). Only SSA identifies the pattern and yields a consistent role mapping. We see that each tool rejects this Decorator instance due to its individual reason - unwillingness to accept deviations or lacking behavioral analyses.

Although the tools disagree on this Decorator instance, it can be reconstructed because three tools strongly agree on the existence of *witnesses* among the program elements in Figure 3:

- Fujaba and Ptidej agree on an instance of State with 80% score from FUJABA and 100% score from Ptidej. Both agree that Command is a State Interface whereas UndoableCommand, CopyCommand,DeleteCommand are Concrete States and CommandButton is a Context.
- SSA and Ptidej agree on an instance of the Command pattern with 25% score from PTIDEJ and 100% score from SSA. Both agree that Command is a Command Interface and also a Receiver whereas UndoableCommand, CopyCommand, DeleteCommand are Concrete Commands and CommandButton is an Invoker.
- PINOT and Fujaba agree on an instance of Strategy with 100% score from PINOT and 80% score from Fujaba. The role mappings are analogous to the ones for State.
- The five role assignments and mappings are complete and consistent.

Command and State are both witnesses of Decorator by virtue of being its superpatterns (see Figures 7, 11, 10). The scores assigned to the recognized subpatterns are high.

We observed that PTIDEJ returns low scores on the deviated pattern instances which are very likely to be true positives (the above Decorator instance got 44% score, the deviated Command witness got 25% score). The authors of PTIDEJ confirmed this. Therefore we do not require PTIDEJ scores to be always high (more than 50%). More generally, the scores of individual tools should be weighted. This is a direction for the future research (see Section 7).

We observed that for correctly identified, deviated pattern instances Ptidej returns low scores (the Command witness got just 25%). Discussions with authors of the analyzed tools (in particular, Yann-Gaël Guéhéneuc and Nikolaos Tsantalis) revealed that different tools have different ways to weigh relationships, classes and variants. So scores from different tools may not be comparable. In the future, we intend to weigh the scores of individual tools in order to compensate their different degree of optimism and pessimism (Section 7).

4.5 Wrongly Identified Superpatterns - Advantages and Drawbacks

If tools identify a superpattern instead of a subpattern, for instance Command instead of Decorator, the overall diagnostic is clearly wrong and prone to confuse users rather than help them. Nevertheless, from a technical point of view, we accept that the superpattern is correctly identified as long as all the criteria for the superpattern are fulfilled (see also Section 3.3.2).

Surprisingly, tools sometimes fail doubly, in that they even fail to correctly identify the superpattern. Different tools may even agree on such wrong diagnoses. For example, for the Decorator instance from Figure 13, PINOT and SSA both classify its State subinstance as a Strategy instance. This occurs because SSA and PINOT use a criterion for distinguishing State and Strategy that fails in this case. [9]

Similarly, it happens that a Command witness for Decorator is sometimes wrongly diagnosed as a Strategy, where Decorator is claimed to be Context and Component Interface is misinterpreted as Strategy Interface. The latter mainly occurs for SSA and Fujaba.

Nevertheless, even wrongly identified superpatterns convey some information about the structure of their superpatterns. For example, the fact that the Decorator pattern is misinterpreted as the Strategy pattern indicates the presence of polymorphism in the Decorator pattern (i.e. performing a particular action depending on the Leaf or Decorator class) - see Section 3.3.2 for more details.

Resilience against such wrong diagnostics is a particular strength of the witness-based approach. Since both, State and Strategy are witnesses for Decorator, it does not matter whether they are distinguished correctly. In general, if the used tools fail to identify one witness, the chance that they will instead identify another small pattern that is a witness too is quite good. In the extreme case they might just identify a tiny EDP, which can still be used by our approach, as explained next. If some witnesses are not supported, more than one EDP can be used to construct this witness.

One should note that wrongly identified superpatterns pose a threat to the data fusion hypothesis. When several tools agree on wrong witnesses, the data fusion approach may produce false positives. We observed that often superpatterns are deviated. The tools that do not support deviations cannot deliver their estimations, making the remaining DPD techniques being fused uncomplementary. We discuss this issue in Section 7.

4.6 EDPs are useful

We note that EDPs are witnesses too, since superpatterns are witnesses and EDPs are just very fine grained superpatterns. The witness-based approach covers the entire spectrum of granularity.

For instance, reconstruction of Observer or Composite needs an instance of the Iterator pattern as a witness[10]. Because the tools analyzed by us do not support Iterator we approximated it by the ArrayMultiReference EDP supported by FUJABA. We used EDPs also to reconstruct strongly deviated Visitor instances. This the demonstrates the flexibility of our approach in taking advantage of whatever witness can be supplied by the used DPD tools.

Another reason for using EDPs is to distinguish patterns with the exactly same set of witnesses (see Table 7). For example, an indication for a Composite is that the iteration is performed on a collection whose elements belong to a supertype of the one that owns the iterating method. Such information is provided by EDPs like Generalization and Association.

On balance, we prefer Strategy, Adapter, Command, State, Template Method and Iterator, which are more reliable witnesses than EDPs since they are bigger and include behavioral aspects. When the tools agree on

[9] In a State instance, PINOT requires the code updating the Current State variable of the Context class to be contained in the Concrete State classes, which is not the case for the Decorator from Figure 13. SSA does not distinguish between State and Strategy, instead reporting the State-Strategy pattern

[10] In Observer, the Subject role uses Iterator to broadcast state changes to Observer instances. Composite uses Iterator to run the action on its children.

such witnesses then we have a much stronger evidence than if they agree on two small EDPs. Thus we can assign the detected instance a much higher score. However, if we need additional evidence in order to distinguish among different similar patterns, we can fall back on suitable EDPs. Section 4.7.2 outlines further research directions.

4.7 More Sources of Information (in addition to witnesses and EDPs)

So far we relied only on the information provided by the evaluated DPD tools. However, there are other easy to use sources of evidence about design patterns, whose input can be fused too. This information can be used:

- To highlight relevant pattern instances when the information about the program structure got from witnesses is not sufficient
- To distinguish between structurally similar patterns after the witnesses have already been collected

In the previous section we already showed how EDPs help to achieve these goals. Now we demonstrate the usefulness of class names and data flow analysis as additional information sources.

4.7.1 Class names carry useful information

As a first experiment we used JTransformer [14], a versatile infrastructure for analysis and transformation of Java programs, to extract information from variable names and comments. Variable names were intensively used in our experiments to find pattern candidates. The names of classes playing the Observer role often contain "Listener" or "Observer". Facade and Mediator class names typically contains "Service", "Repository" etc. In the Visitor pattern, the classes playing Visitor role are typically named as "...Visitor" - for example, JavaParserVisitor. The information got from class names was especially valuable in spotting pattern deviations when the information got from witnesses was vague (for example, the class names were useful in detecting 1:1 Observers - particularly, in distinguishing them from Adapter or Bridge instances).

4.7.2 Navigating within the group of structurally similar patterns

From Figure 7 we note that, after the witnesses are collected, the pattern to be detected is expected to reside within the group of structurally similar patterns. Additional information (EDPs, data flow, names) may be needed to navigate between the patterns in the same group. In Section 4.6 we already used the Inheritance EDP was used to distinguish between Composite and Observer.

In our experiments, the class and method names were used as well - for example, in most cases the names of the classes playing the Observer role ended with the word "listener". This information was useful for distinguishing between Observer and Visitor.

Data flow and control flow information can be used to distinguish between the patterns having the same set of witnesses. For example, the Decorator, Chain of Responsibility and Proxy patterns have the same set of witnesses, namely Command and State. To distinguish between Proxy and Decorator, data flow may be used (if the Component subclass performs delegation only to some concrete sibling, we have Proxy). To distinguish Chain of Responsibility from Decorator we need control flow information (Chain of Responsibility delegates to the next subclass of Component interface under certain conditions and not ALWAYS, as Decorator).

5 Evaluation of the Data Fusion Approach

We verified the data fusion hypothesis manually, strictly following the conceptual framework laid out in the previous section. For our evaluation we reviewed several tens of examples from the repositories listed in Table 3,

		Detected Instances						True Instances
		Jointly recognized		Reconstructed		Total Detected by Data Fusion		
		true	false	true	false	true	false	
Patterns	Decorator (Proxy, COR)	3	0	7	3	10	3	10
	Visitor	2	0	6	3	8	3	10
	Observer (Composite)	2	0	4	2	6	2	10
	Bridge	2	0	0	12	2	12	5
	Mediator (Façade)	5	0	0	25	5	25	5

Table 5. Results from the manual application of data fusion.

		Instances Detected By Individual Tools									
		FUJABA		PTIDEJ		PINOT		DP-Miner		Similarity Scoring	
		true	false	true	false	true	false	true	false	true	false
Patterns	Decorator (Proxy, COR)	---	---	8	13	6	4	---	---	8	6
	Visitor	5	10	5	12	3	3	---	---	6	4
	Observer (Composite)	5	12	5	11	3	4	---	---	6	5
	Bridge	2	14	---	---	2	12	2	10	---	---
	Mediator (Façade)	---	---	---	---	5	25	5	15	---	---

Table 6. Instances detected by individual tools

trying to detect Observer, Visitor, Decorator, Chain of Responsibility, Proxy, Bridge, Mediator and Facade instances.

Table 5 presents the results. It lists how many instances were correctly or wrongly recognized or reconstructed for each of the different patterns. For comparison, the right-most column shows the numbers of true instances that should have been identified. One can see the improvement by comparing this table with the results returned by individual tools (see Table 6).

The design patterns in the first, third and fifth row are grouped together because they have the same set of witnesses. In these cases we fall back to the use of EDPs as additional witnesses, as described in Section 4.6. For simplicity, we take just one pattern from each group as a representative in the following.

Improved precision and recall The first three rows of Table 5 illustrate cases where witness-based fusion worked very well. In our experiment, we were able to recognize or reconstruct all 10 Decorators (CoRs, Proxies), 8 of 10 Visitors and 6 of 10 Observers that we had chosen as our benchmarks. Some very deviated instances were not reconstructed (1 : 1 Observer or Visitors without double dispatch that occur sometimes in Eclipse source code).

		Precision					
		FUJABA	PTIDEJ	PINOT	DP-Miner	Similarity Scoring	Fusion
Patterns	Decorator (Proxy, COR)	---	38%	60%	---	57%	77%
	Visitor	30%	30%	50%	---	60%	73%
	Observer (Composite)	29%	31%	43%	---	55%	75%
	Bridge	12%	---	14%	17%	---	0%
	Mediator (Façade)	---	---	20%	25%	---	0%

Table 7. Precision: Individual tools versus data fusion

		Recall					
		FUJABA	PTIDEJ	PINOT	DP-Miner	Similarity Scoring	Fusion
Patterns	Decorator (Proxy, COR)	---	80%	60%	---	80%	100%
	Visitor	50%	50%	30%	---	60%	80%
	Observer (Composite)	50%	50%	30%	---	60%	60%
	Bridge	40%	---	40%	40%	---	0%
	Mediator (Façade)	0	---	100%	100%	---	0%

Table 8. Recall: Individual tools versus data fusion

Any individual tool detected at most 8 Decorators (Ptidej , SSA), 6 Visitors (SSA) and 6 Observers (SSA) (see Table 6). Thus we could improve recall. It should be noted that SSA applies very liberal matching policy in order to accept deviated instances; therefore it produces better recall than other tools. Table 8 shows the recall of each analysed tool as a percentage and compares it to the recall achieved by the fusion approach. Ptidej applies liberal matching crireria as well. The recall produced by PINOT is low due to restrictive detection policies.

For the Decorator, Visitor and Observer, precision could also be improved significantly compared to any individual tool. Whereas fusion achieved precision of 73% to 77% for Decorator, Visitor and Observer, the precision of the analyzed tools ranged on the same patterns between 30% and 60% (see Table 7). PINOT achieves high precision because it applies very restrictive matching policies. False alarms generated by SSA for the Decorator-Proxy-CoR group occur because SSA lacks strong behavioral analyses (for example, classifying CoR instances as Decorators). To detect Visitor and Observer, SSA uses class names in order to filter out false alarms .

It is not surprising that we were able to increase precision as well as recall, given that each reconstructed pattern removes a false negative and several false positives (its witnesses) from the output set of the basic DPD tools.

Remaining false positives The remaining ratio of false positives is explained by the fact that tools may agree on wrong witnesses. It obviously does not count how many tools agree but how complementary the techniques used by the agreeing tools are. If they use the same techniques, they are often fail on the same cases. We will use this insight in the future by assigning to each detected instance a confidence score based on the scores assigned by the tools and on a "synergy function" that assigns high values to agreements of tools that employ complementary techniques.

Need for improvements of tools Our approach failed on the analysed instances of Bridge, Mediator and Facade since there were many incorrectly recognized instances of these patterns in the results of the employed tools. For instance, Fujaba, Pinot and DP-Miner return false instances of Bridge, which are actually deviated Observers (when only one Observer exists for the Subject). Failures occurred mainly because the analysed tools did not recognize indirection-based deviations. We observed that tools usually report Facade when a group of classes G invokes some service at a class F, which, in turn, uses auxiliary classes $S_{1,...}S_{\mathrm{n}}$ to complete its job. This approach brings many false positives (around 85% of the total number of candidates). For example, the class org.jhotdraw.figures.PolyLineFigure from JHotDraw 6.0 draws the polygon. It uses the class Collections-Factory to initialize the set of points for redrawing, the class org.jhotdraw.util.Geom to compute the redrawing rectangle and the classes org.jhotdraw.util.StorableInput (StorableOutput) to load the information about the corner points from the file.

Obviously, data fusion does not replace work on further improvement of existing tools. The main areas to be addressed are dataflow analyses, support for indirection and implementation of additional pattern detectors, such as the missing detector for Iterator discussed above. Name-based criteria (see also [27]) also seem to be promising according to our evaluation. Data fusion will be able to take advantage of any improvements of the analysed tools

	Class	Inheri-tance	Field	Method	Method call	Field access
SSA	✓	-	-	-	-	-
DP-Miner	✓	-	-	-	-	-
PINOT	✓	✓	✓	✓	✓	✓
PTIDEJ	✓	-	-	-	-	-
FUJABA	✓	-	-	✓	-	-

Table 9. Roles reported by the different tools.

or the availability of new tools.

Need for a uniform exchange format The biggest problems for fusion turned out to be that most tools report just a fraction of the information that they derived internally. In particular, reconstruction requires witnesses to completely cover the mandatory roles of of the reconstructed pattern (including methods and fields, if necessary). This is not possible if tools only report roles at class level, as done by SSA, DP-Miner and Ptidej. Fujaba additionally reports roles at method level. PINOT reports the entire range of roles, down to the granularity fields, methods, individual method calls and field accesses (see Table 9). Also, different tools do not assign qualitative labels ($High, Low$) to their scores, making them incomparable. For example, SSA pattern detection approach clearly states that the pattern candidates scored lower than 50% should be rejected. On the other hand, Decorator instances scored with 44% should be accepted (based on the conversations with the authors).

Therefore, we identified the need for common meta-model and a uniform exchange format for DPD tools. We invited the authors of the reviewed tools to collaborate on this issue and received very positive feedback. We expect that significantly better results can be achieved if tools provide finer-grained role mappings and other possibly relevant information that they hold.

Lacking information, we would have either been forced to weaken our detection criteria or to accept worse results in certain cases. In particular, reconstruction requires witnesses to completely cover the mandatory roles of of the reconstructed pattern. This is not possible if tools do not report roles beyond the granularity of classes. Therefore, we claim the need for a uniform exchange format for DPD tools, based on a common metamodel of patterns and roles. We invited the authors of the reviewed tools to colaborate on this issue and received very posititve feedback. We expect that significantly better results can be achieved if tools provide more fine grained role mappings and other possibly relevant information that they derive.

Conclusions We conclude that the data-fusion-based approach to DPD recommends itself as a good way to build on the substantial expertise already available in existing DPD tools and to improve detection quality beyond the level achievable currently by any single tool. The list of pattern candidates resulting from applying the data fusion hypothesis appears to be more valuable than the result of any individual recognition tool for a promising number of cases. In the cases where the approach failed, we identified improvements that will overcome the limitations of our very first experiments reported in this paper.

6 Related Work

Data Fusion in Software Engineering To the best of our knowledge, our approach is the first one that investigated an approach to design pattern detection based on data fusion. However, the idea of applying data fusion in software engineering is not entirely new. For instance, Poshyvanyk et.al [21] suggest using data fusion to detect

features in big programs (for example, for bug finding). They suggest to combine the outputs of the static-analysis-based and the dynamic-analysis-based concept detection tools. That paper partly inspired our approach. We went one step further by combining the output of tools that already combine different techniques.

State of the art overviews In the previous state of art overview by Dong and Peng [7], the comparison of tools was mostly based on literature, without running the tools. Our contribution in this domain is that we performed extensive practical evaluation of five tools on various code repositories that cover a wide range of relevant scenarios. Our experimentation enabled us to provide detailed feedback to the various tool authors, regarding the robustness, performance, scalability, precision and recall of the approaches. The authors of another state-of-the-art review (see [9]) did similar work, running 3 DPD tools (analyzing C++ programs) on an extensive set of program repositories. The authors made several conclusions why different tools render different results on the same code and are currently striving towards a common benchmark for DPD tools.

Relationships among patterns The decomposition of patterns into smaller subparts (namely EDPs) has already been suggested in SPQR [26, 24, 25] and implemented in SPQR, FUJABA [17][17], EDPDetector4Java[3] and to a limted extent (one EDP) in SSA [28]. We take this decomposition one step further by showing that also much bigger entities can beneficially be used as witnesses of sought patterns. Our witness relation was based on our subpattern relation. However, other relations between patterns have been investigated by Zimmer [32]and could be integrated into our framework in the future.

Mandatory and optional roles To the best of our knowledge, several approaches touch the idea of optional and mandatory roles. For instance, SSA outputs only the two most important roles since it is easy then to identify the actual pattern instance. This was done for the sake of simplicity. Liberal role checking performed in PTIDEJ [1] allows some roles which are considered to be less important to be missed so it is similar to our concept. Di Penta et.al [19] refers to this concept as "main roles". The difference between the above approaches and ours is that we emphasized that roles need to be regarded at any granularity, not just at the level of classes, as done in many tools.

Two-Staged Approach Our approach is partly based on performing two analysis stages. In the first one, we identify groups of structurally similar patterns that have the same witnesses. In the second one, we use additional information (e.g. EDPs etc - see Section 4.7) to distinguish between the patterns within the same group. This is similar to the approach of Lucia et al. [5], where design pattern instances are first identified by considering design structure (derived from specifications expressed by means of a visual language). For example, when a class holds a "parent reference" to the object belonging to its superclass, one has a Decorator candidate. One can improve the results at the second stage by performing a fine-grained source code analysis (for example, Decorator candidates who do not have a method delegating the control to its superclass via the "parent reference" will be rejected). The delegator and the delegatee must have identical signatures. Lucia et al., however, do not use data fusion from different tools.

Striving Towards A Common BenchMark for evaluation of DPD tools To the best of our knowledge, the demand for a solution to evaluate patterns effectively and easily started to appear in conferences in 2006 (see Petterson et al [20], also visit their site `http://w3.msi.vxu.se/~npe/DPDES/`). Using this work as a starter, Fulop et al.[9] evaluated several DPD tools (in C++) and built the common benchmark.

7 Conclusions and Future Work

In this paper we reported on the findings obtained by a practical evaluation of DPD tools on a large variety of benchmarks and described how they form a novel approach to design pattern detection based on data fusion. We

introduced the concept of sub- and superpattens and used superpatterns as witnesses for bigger patterns on which the evaluated tools failed. The introduced techniques of joint recognition and reconstruction of patterns from witnesses were shown to improve precision and recall for the analysed instances of Decorator, Proxy, Chain of Responsibility, Visitor, Observer, and Composite beyond the level achieved by any of the analysed tools. In the cases where the approach failed, we identified improvements that will overcome the limitations of our first experiments reported in this paper. These improvements address the fusion method itself as well as the current functional limitations of existing tools. In addition, we showed that support of a common export format by the tools would significantly enhance the power of data fusion. The main contribution of our paper to the state of the art is having demonstrated that data fusion is an effective and promising approach that should be further pursued in the future. So far we performed our experiments manually. Next we will refine our approach based on the insights from this paper and implement it, collaborating in parallel with the authors of existing DPD tools on the design and implementation of a common DPD interchange format. Last but not least, we will include emerging new tools, such as MoDeC [16] in future experiments.

We noted that often witnesses are themselves deviated. Therefore, it is important that the DPD tools that detect witnesses support deviations. If they do not, the number of tools that agree on a certain witness might be too low to provide sufficient evidence. In addition, there is the risk that the remaining tools that agree incorporate similar techniques and therefore yield the same false positives or false negatives. For example, both analysed DPD tools that apply static behavioural ananlyses, PINOT and DPminer, do not support deviations at all. Therefore, they often do not participate in agreements. Among the remaining tools Ptidej and Fujaba incorporate similar (fine-grained static structural) techniques. Therefore, their agreement is less trustworthy. Agreement is most valuable if the tools apply complementary techniques. Taking this properly into account is an issue for future investigation.

8 Acknowledgements

We are thankful to Yann-Gaël Guéhéneuc, Lothar Wendehals, Nikolaos Tsantalis, Clement Izurieta and Jim Bieman for fruitful discussions. We also thank Yann-Gaël Guéhéneuc, Lothar Wendehals, Nikolaos Tsantalis and Nija Shi for providing us with their DPD tools, helping to install and to use them. Last but not least, we thank Daniel Speicher, Holger Mügge and Jörg Westheide for a lot of useful feedback and discussions. They also spotted some important omissions and confusing wording in a preliminary version of this paper.

A Appendix : Decorator, Proxy, and Chain of Responsibility: True Positives

1. 3 Ideal instances (all from $java.io$; all analyzed tools agree on them):
   (a) Decorator: java.io.InputStream as Component, java.io.BufferedInputStream as Decorator.
   Optional Roles: FileInputStream, ObjectInputStream as Leaves
   (b) Decorator: java.io.OutputStream as Component, java.io.OutputStream as Decorator.
   Optional Roles: FileOutputStream, ObjectOutputStream as Leaves
   (c) Decorator: java.io.Reader as Component, java.io.BufferedReader as Decorator.
   Optional Roles: FileReader, PipedReader as Leaves

2. 5 Decorator & Proxy & Chain of Responsibility instances (reconstructed from witnesses):
   (a) Decorator (JUnit 3.7) : junit.framework.Test as Component,
   junit.extensions.TestDecorator as Decorator
   Optional Roles : junit.exetnsions.TestCase as Leaf;
   junit.extensions.RepeatedTest, junit.extensions.TestSetup as Concrete Decorators
   Deviations: Decorator is not abstract; Leaf is the interface (to be implemented by users)

(b) Decorator (JHotDraw 6.0): org.jhotdraw.framework.Command as Component,
org.jhotdraw.standard.UndoableCommand as Decorator
Optional Roles : org.jhotdraw.standard.CopyCommand, DeleteCommand as Leaves;
Deviations: Decorator is not abstract; Decorator-Component delegation is made indirectly (through the getter)

(c) Chain of Responsibility (Eclipse runtime core):
org.eclipse.update.internal.core.InstallHandlerProxy as ConcreteHandler,
org.eclipse.update.core.IInstallHandler as Handler

(d) Proxy (Nutch 0.4): org.apache.nutch.searcher.Searcher as Component
org.apache.nutch.searcher.NutchBean as Proxy
org.apache.nutch.searcher.IndexSearcher as Real Subject
Delegating Method :NutchBean.getExplanation(), NutchBean.search()

(e) Chain of Responsibility (Eclipse OSGI):
osgi.framework.FilteredServiceListener as ConcreteHandler,
osgi.framework.ServiceListener as Handler

3. 2 Decorator and Proxy instances (too deviated; were not reconstructed from witnesses)

(a) Decorator (JHotDraw 6.0): org.jhotdraw.framework.FigureEnumeration as Component,
org.jhotdraw.standard.FigureAndEnumerator as Decorator
Action method: nextFigure()
Deviation: Standard Decorator holds only one Component reference the action is delegated through. In this instance two FigureEnumeration references are needed (to implement AND enumeration). DPD tools usually report only one of these references.

(b) Proxy (JHotDraw 6.0): org.jhotdraw.framework.Tool as Component,
org.jhotdraw.contrib.zoom.ZoomTool as Proxy
Optional roles : org.jhotdraw.contrib.zoom.ZoomAreaTracker as Concrete Subject
Deviation: Proxy is initialized in the mouseDown method that,
according to the concrete application logic, is called immediately after the constructor.

B Appendix : Observer and Composite: True Positives

1. 2 Ideal instances (all analyzed tools agree on them):

(a) Observer (JUnit 3.7): junit.framework.TestListener as Observer, junit.framework.TestResult as Subject

(b) Composite (Quick UML 2001): diagram.tool.Tool as Component; diagram.tool.CompositeTool as Composite

2. 4 Observer instances (reconstructed from witnesses):

(a) Observer (JHotDraw 6.0): org.jhotdraw.framework.ViewChangeListener as Observer,
org.jhotdraw.application.DrawApplication as Subject

(b) Observer (PMD 2.8): net.sourceforge.pmd.cpd.CPDListener as Observer,
net.sourceforge.pmd.cpd.MatchAlgorithm as Subject
Note : 1 : 1 Observer, detected by using class names (including “Listener”).

(c) Observer (Quick UML 2001): diagram.tool.ToolListener as Observer, diagram.tool.AbstractTool as Subject
Note : 1 : 1 Observer, detected by using class names (including "Listener").

(d) Observer (PMD 2.8): net.sourceforge.pmd.ReportListener as Observer, net.sourceforge.pmd.Report as Subject
Note : 1 : 1 Observer, detected by using class names (including "Listener").

3. 4 Observers & Composites (were not currently reconstructed from witnesses)

(a) Observer (JHotDraw 5.1): CH.ifa.draw.util.CommandMenu as Subject, java.awt.MenuItem as Observer
Trick: CommandMenu maintains the collection of Command objects but notifies MenuItem objects associated with these Commands and not Commands themselves. The iteration is performed over the collection of MenuItem objects that resides inside AWT modules.

(b) Observer (PMD 2.8): org.acm.seguin.parser.ast.ASTClassDeclaration as Subject, org.acm.seguin.pretty.JavaDocableImpl as Observer
Trick: ASTClassDeclaration.finish() makes sure that all java docs are present by calling JavaDocableImpl jdi.require (classtag...). Though Subject and Observer are in the same hierarchy (they could be seen as Proxy and Real Subject), ASTClassDeclaration may update JavaDoc holders of any changes. Therefore we think this is an 1:1 Observer where comments carry useful information for revealing this instance.

(c) Observer (QuickUML 2001): uml.ui.Tool as Observer, uml.ui.ToolPalette as Subject (notifier: $ToolPalette.propertyChange(...)$ - we noticed this naming convention later and therefore did not have enough information to spot this 1:1 Observer)

(d) Composite (JHotDraw 6.0): org.jhotdraw.contrib.QuadTree as Component and Composite, QuadTree.clear() is the action
This is a very deviated Composite. Instead of holding a collection of children, QuadTree holds four fields
(East, West, North, South) to model four subpanels. Redrawing is called on all subpanels recursively.

C Appendix : Visitor: True Positives

1. 2 Ideal instances (tools agree on them):

(a) (PMD): net.sourceforge.pmd.ast.JavaParserVisitor as Visitor, net.sourceforge.pmd.ast.ASTExpression as Element

(b) (JRefactory 2.6) : org.acm.seguin.summary.SummaryVisitor; org.acm.seguin.summary.FieldSummary as Element

2. 6 Visitor instances to be reconstructed (deviated; recognized by at least one tool; reconstructed from witnesses):

(a) (JHotDraw 6.0): org.jhotdraw.framework.FigureChangeListener as Element, org.jhotdraw.framework.FigureVisitor as Visitor

(b) (JHotDraw 6.0): org.jhotdraw.framework.Figure as Element, org.jhotdraw.framework.FigureVisitor as Visitor

(c) (Eclipse CVS core): org.eclipse.team.internal.ccvs.core.resources.ICVSFolder, ICVSFile as Element;
org.eclipse.team.internal.ccvs.core.resources.ICVSResourceVisitor as Visitor

(d) (Eclipse Runtime core): org.eclipse.core.internal.events.ResourceDelta as Element;
org.eclipse.core.resources.IResourceDeltaVisitor as Visitor

(e) (Eclipse Runtime Core) org.eclipse.core.internal.watson.ElementTreeIterator as Element;
org.eclipse.core.internal.watson.IElementContentVisitor as Visitor

(f) (Eclipse core) : org.eclipse.core.internal.resources.mapping.ProposedResourceDelta as Element;
org.eclipse.core.resources.IResourceDeltaVisitor as Visitor.

3. 2 very deviated Visitors (not recognized by any tool):

(a) (Eclipse runtime core): org.eclipse.core.resource.mapping.ResourceTraversal as Element;
org.eclipse.core.resource.IResourceVisitor as Visitor
Deviation: ResourceTraversal.accept(IResourceVisitor) is the correct accept method. But it does not call
directly visit(IResourceVisitor). It only calls IResource.accept() for each resource being owned. There is no double dispatch in this ConcreteElement. Note that there are Concrete Elements with double dispatch and false
Visitors that contain double dispatch! (see the list of data fusion false positives).

(b) (JRefactory 2.6): org.acm.seguin.parser.ast.SimpleNode as Element;
org.acm.seguin.parser.JavaParserVisitor as Visitor;
Deviation: SimpleNode.childrenAccept() is the correct accept method. But it does not call
directly visit(). It only calls jjtAccept(). Reconstruction from witnesses reports
SimpleNode.jjtAccept() as accept method, which is wrong (not the whole structure was traversed).

D Appendix : Bridges: True Positives

1. 2 Ideal instances (tools agree on them):

(a) (Nutch 0.4): net.nutch.db.DistributedWebDBReader.EnumCall as Abstraction;
java.util.Enumeration as Implementor
Concrete Abstraction 1: net.nutch.db.DistributedWebDBReader.LinkEnumCall
Concrete Abstraction 2: net.nutch.db.DistributedWebDBReader.PageEnumCall
Concrete Abstraction 3 : net.nutch.db.DistributedWebDBReader.PageByMD5EnumCall
Concrete Implementor 1 : net.nutch.db.DBSectionReader.MapEnumerator
Concrete Implementor 2 : net.nutch.db.DBSectionReader.IndexEnumerator
Concrete Implementor 3 : net.nutch.db.DBSectionReader.TableEnumerator

(b) (Java AWT) : java.awt.Container as Abstraction;
java.awt.LayoutManager as Implementor
Concrete Abstractions: Pane, ScrollPane, EditorPane;
Concrete Implementors: BorderLayout, FlowLayout

E Appendix : Mediator and Facade: True Positives (all tools agree on them)

1. Facade (JHotDraw 6.0) : java.awt.Container as Abstraction
   org.jhotdraw.util.UndoManager is Facade ,
   org.jhotdraw.util.Undoable as subsystem classes

2. Facade (JHotDraw 6.0): org.jhotdraw.util.StorageFormatManager as Facade ,
   org.jhotdraw.util.StorageFormat classes as subsystem classes

3. Facade (JHotDraw 6.0): org.jhotdraw.contrib.html.StandardDisposableResourceManager as Facade,
   org.jhotdraw.contrib.ResourceHolder subclasses as subsystem classes
   (DisposableResourceHolder, StandardDisposableResourceHolder)

4. Mediator (Java AWT): java.awt.Container as Mediator class,
   Colleagues are java.awt.Component subclasses
   (Choice, TextComponent , Scrollbar , Checkbox , List , Label , Panel , TextArea)

5. Mediator (Java AWT): DefaultKeyboardFocusManager as Mediator class,
   Colleagues : java.awt.KeyEventDispatcher, KeyEventPostProcessor
   Colleagues are referenced through lists keyEventDispatchers, keyEventPostProcessors
   A key event is generated by an AWT entity and sent to the dispatcher by means of calling KeyEventDispatcher.dispatchEvent. Then it is rerouted to an appropriate key event processor (routing scheme is within Mediator class)

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *ASE'01*, page 166, San Diego, USA, Nov. 2001. IEEE Computer Society.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *IWPC'98*, page 153, Washington, USA, 1998. IEEE Computer Society.

[3] F. Arcelli, S. Masiero, and C. Raibulet. Elemental design patterns recognition in java. *STEP'05*, 0:196–205, 2005.

[4] B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice*. Addison Wesley, 2009.

[5] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi. A two phase approach to design pattern recovery. In *CSMR '07*, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society.

[6] J. Dong, D. S. Lad, and Y. Zhao. Dp-miner: Design pattern discovery using matrix. In *ECBS'07*, pages 371–380, Washington, USA, 2007. IEEE Computer Society.

[7] J. Dong, Y. Zhao, and T. Peng. A review of design pattern mining techniques. *IJSEKE*, 2008.

[8] R. Ferenc, A. Beszedes, L. Fülöp, and J. Lele. Design pattern mining enhanced by machine learning. In *ICSM'05*, pages 295–304, Washington, USA, 2005. IEEE Computer Society.

[9] L. J. Fülöp, A. Ilia, A. Z. Vegh, and R. Ferenc. Comparing and evaluating design pattern mining tools. In *Proceedings of SPLST '07*, 14th June 2007.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1994.

[11] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *IJCAI'01*, pages 57–64, Seattle, USA, Aug. 2001. AAAI Press.

[12] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. *WCRE*, 0:172–181, 2004.

[13] O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *CSMR'06*, pages 175–184, Washington, USA, 2006. IEEE Computer Society.

[14] G. Kniesel, J. Hannemann, and T. Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of LATE '07*, New York, USA, 12th March 2007. ACM.

[15] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. In *COMPSAC*, pages 574–579, 2001.

[16] J. K.-Y. Ng, Y.-G. Guéhéneuc, and G. Antoniol. Identification of behavioral and creational design patterns through dynamic analysis. *JSME*, 0(0):0, 2009. submitted.

[17] J. Niere, J. P. Wadsack, and L. Wendehals. Handling large search space in pattern-based reverse engineering. In *IWPC'03*, page 274, Washington, USA, 2003. IEEE Computer Society.

[18] J. Odrowski and P. Sogaard. Pattern integration - variations of state. In *Proceedings of PLoP 1996*, 6th September 1996.

[19] M. D. Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *ICSM*, pages 217–226. IEEE, 2008.

[20] N. Pettersson, W. Löwe, and J. Nivre. On evaluation of accuracy in pattern detection. In *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, Oct. 2006.

[21] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE TSE*, 33(6):420–432, 2007.

[22] D. Riehle. Composite design patterns. In *OOPSLA'97*, pages 218–228. ACM Press, 1997.

[23] N. Shi and R. A. Olsson. Reverse engineering of design patterns from java source code. In *ASE'06*, pages 123–134, Washington, USA, 2006. IEEE Computer Society.

[24] J. Smith and D. Stotts. An elemental design pattern catalog. Technical Report tr02-040, The University of North Carolina, CS Department, Dec. 2003.

[25] J. Smith and D. Stotts. Elemental design patterns and the rho-calculus: Foundations for automated design pattern detection in spqr. Technical Report tr03-032, The University of North Carolina, CS Department, Sept. 2003.

[26] J. Smith and D. Stotts. Spqr: Flexible automated design pattern extraction from source code. In *ASE'03*. IEEE, 2003.

[27] P. Tonella and G. Antoniol. Object oriented design pattern inference. In *ICSM'99*, page 230, Washington, USA, 1999. IEEE Computer Society.

[28] N. Tsantalis and S. T. Halkidis. Design pattern detection using similarity scoring. *IEEE TSE*, 32(11):896–909, 2006.

[29] L. Wendehals. Improving design pattern instance recognition by dynamic analysis. In *WODA'03*, Portland, USA, 2003. IEEE Computer Society.

[30] L. Wendehals. *Struktur- und Verhaltensbasierte Entwurfsmustererkennung*. Phd thesis, Universität Paderborn, Institut für Informatik, September 2007.

[31] L. Wendehals and A. Orso. Recognizing behavioral patterns at runtime using finite automata. In *WODA'06*, pages 33–40, New York, USA, 2006. ACM.

[32] W. Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design*, pages 345–364. Addison-Wesley, 1995.