Standing on the Shoulders of Giants – 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 on various Java projects. 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 can be a* witness *for the existence of another pattern. Our approach is able (1) to provide correct diagnostics even if the inputs from the evaluated tools were partly wrong and (2) to detect patterns instances not identified by the individual tools. 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. We also found that on the analyzed 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 [6] are domain-independent descriptions of groups of communicating classes that present solutions to some 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 candidates) [5]. 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* [5]. Precision and recall are conflicting issues. 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 known 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 by Dong and Peng [5] and by our own practical evaluation of five existing tools (Section 3).

We claim that, instead of striving for the ideal DPD tool it is more promising to combine the decisions 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. In order to verify our claim, we evaluated several pattern detection tools and developed the *witness-based approach* for combining their results. Our contributions include

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

The next section introduces basic notions. Section 3 describes our practical evaluation of five existing tools. Section 4 describes our approach to fusion-based design pattern detection. Section 5 evaluates it and outlines future research directions. Section 6 discusses related work not touched previously. Section 7 summarizes our contributions. For details that exceeded the page limit of this paper see [9].

2 Background

A *design pattern* description comprises a set of *roles*. Roles are duties that can be fulfilled by program elements (types, methods etc.), relations (inheritance, association etc.) or collaborations (expressed by fragments of code or of dynamic UML diagrams). For example, in the Decorator pattern [6] (Figure 1), the type known to clients plays the Component role . An *instance* of a design pattern *P* (called *design motif* in [7]) is a set of program elements that together play some roles of *P*. A pattern *candidate* reported by a DPD is a set of program elements that might be an instance.

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 the classes Component and Decorator, their subtype relation, the forwarding method and its contained invocation of the same method on the component are mandatory. They implement the basic concept of the pattern. The roles Concrete Component and Concrete Decorator are optional because their implementation may be deferred.

Original pattern descriptions are rather informal, so programmers need to adapt them to specific situations [6]. 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* [11, 18, 5], 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 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. [16] and Kaczor et al. [8] provide examples of indirection. The opposite of indirection is *role merging*, which occurs if the same program element plays different roles in the same pattern (see [5]). Property relaxation, indirection, role merging and the lack of some optional roles are the basic deviations that we regard in this paper. Deviations are one of the main challenges for DPD approaches.

[1] Guéhéneuc et al. [7] call them *distorted micro-architectures*.

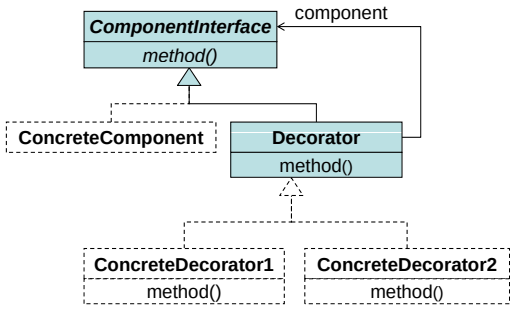

Figure 1. The Decorator pattern. Mandatory roles are indicated by plain lines and dark background, optional ones by dashed lines and white background.

3 Tool Evaluation

In this section we explain our selection of tools and benchmarks, give a short introduction to each tool and explain the findings that motivated the fusion-based approach and contributed to the main solution idea.

3.1 Evaluated Tools

Among the 14 DPD approaches for Java listed by Dong et al. [5], we found that 3 have no accessible prototype, 2 are not freely available and one is implemented just for Linux, leaving only 8 candidates for practical experimentation. From them, we selected the 5 tools listed in Table 1. Taken together, they cover all currently known basic pattern analysis approaches[2]. By *basic approaches* we mean elementary, static or dynamic analyzes of program structure or behavior [9]. Among the candidates, we favored tools that are able to detect pattern variations:

- SSA (Similarity Scoring) [18] only performs static structural analyzes of inter-class relationships, method signatures and method calls. SSA is able to tolerate

[2] The sole exception are techniques supported only by the unavailable SPQR tool [16].

	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 1. Analyzed tools and the techniques applied by them

small pattern deviations (up to one edge in the program graph).

- DP-Miner [4] performs thorough static behavioral analyzes. It does not support pattern deviations. No dynamic analyzes are performed.
- PINOT [14] performs thorough static behavioral analyzes. It does not support any variations. No dynamic analyzes are performed.
- Ptidej [7] 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 analyzes are limited to the identification of object creation.
- Fujaba [11] decomposes patterns into EDPs (often called "subpatterns" in Fujaba), which enables recognition of deviations, including cases of role merging and attribute relaxation. Its successor, Reclipse, based on the PhD work of Wendehals [20, 21, 19], 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 practical evaluation of the above tools on several tens of design pattern instances. Part of the actual instances were taken from PMart[3]. Others were identified by us manually. The analyzed repositories cover a wide range of relevant scenarios. Lexi 0.1, JUnit 3.7, JHotDraw 5.1 and 6.0 and QuickUML 2001 are small libraries containing only a few variations[4]. PMD 1.8 and JRefactory 2.6 and 2.8 are bigger frameworks which are used in the real scientific and industrial applications. Apache 4.1.37 and Eclipse platform 3.1 code are examples of large, quickly evolving industrial applications containing many pattern deviations. This wide range was essential for realistic results since, according to our observations, tools that perform well on "clean" code can perform badly on the code containing many deviated pattern instances[5].

3.3 Findings

Our experiments confirmed three main claims of Dong et al. [5] that in esence motivated our investigation of data fusion: (1) No existing tool implements all known basic DPD approaches and (2) different pattern detection tools render different results on the same code. In addition, we made several observations that pointed us to the criteria on which data fusion 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 candidates as candidates of more general patterns. For example, Ptidej and Fujaba often classify Decorator candidates as State candidates. Figure 2 illustrates such a case. It shows the classes of a Decorator instance from JHotDraw 6.0 and their inferred mapping to the roles of theState pattern. This instance is not recongnized as a Decorator because of property relaxation: the class AbstractCommand playing the Leaf role is abstract (according to [6], it should be concrete).

Our experiments yielded more misclassifications of the same type, which occur for similar instances. We conjecture that they are caused by the fact that for some variations only a subset of the constraints defining the pattern holds. If there is a pattern that is characterized by this weaker set of constraints, it will be wrongly detected instead of the correct one.

[3]The authors thank Yann-Gaël Guéhéneuc for his help and discussions on PMart `http://ptidej.dyndns.org/downloads/pmart/`.

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

[5]It was also essential for evaluating speed and scalability. However, we do not address these issues here. See [9] for details.

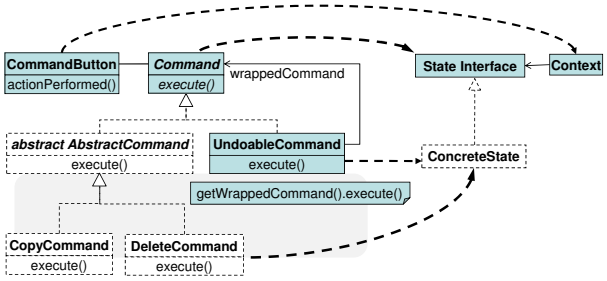

Figure 2. Decorator instance classified as State by Ptidej and Fujaba

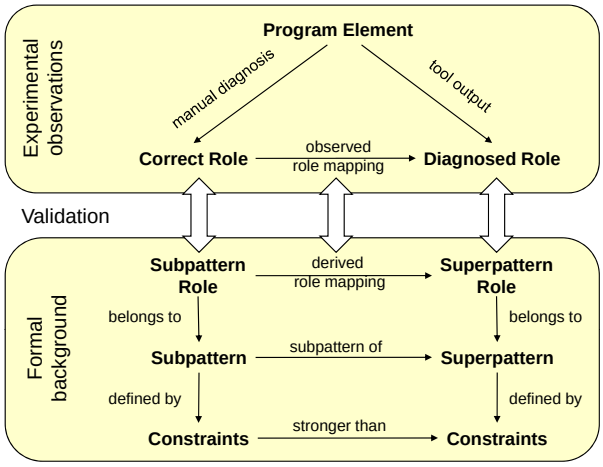

Figure 3. Validation of role mappings.

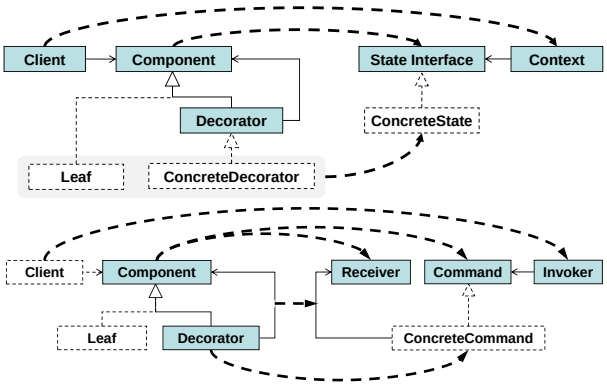

Figure 4. Subpattern role mapping from Decorator to State and to Command

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 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 (more instances).

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 behavior change.

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 [6] and compared the results to the ones obtained experimentally on the benchmarks described in Section 3.2. First, we specified roles and defining constraints for all the analyzed patterns. From these constraints we derived 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[6]. 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 3 gives an overview of the described validation method.

Figure 4 illustrates the subpattern relation of Decorator to State and to Command, including the related role mappings. The role mappings are also presented textually in the first and second row of Table 2 along with the constraints that are strengthened in the subpattern. An example of a strengthened constraint is contained in the first row of the table: the Command and Receiver roles must coincide so that the Component role can be mapped to both. The rows 3 and 4 show that subpattern role mappings may not be unique, if different roles are defined by the same con-

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

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 → Command	**Visitor** → Command, **Element** → Invoker=Receiver
Visitor → Command	**Element** → Command, **Visitor** → Invoker=Receiver

Table 2. Subpatterns and role mappings for several big design patterns

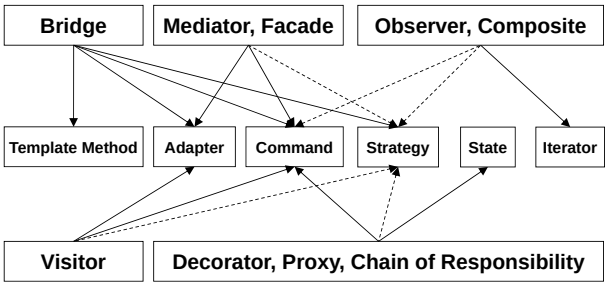

Figure 5. Subpattern relations.

straints. In the Visitor pattern, the Visitor and the Element mutually call each other (double dispatch). Thus they can both be interpreted as playing the role of Command.

Figure 5 illustrates all observed subpattern relations. It shows that, at a technical level, 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. Some patterns have the same superpatterns. However, their role mappings differ. The solid lines indicate observed subpattern relations that coincide with the formally derived ones. The dashed lines indicate observed subpattern relations that resulted from false positives produced by the analysed tools.

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 weaker set of constraints - for example, replacing composition by association or completely eliminating some constraints. A smaller pattern does not necessarily need to be a superpattern. We observed that

- the number of true positives among smaller patterns (Adapter, Command, State, etc.) is much higher than among bigger ones (Decorator, Mediator etc.).
- the analyzed 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 weaker constraints must be satisfied. Furthermore, smaller design patterns are conceptually simpler and give rise to less variations. This is consistent with the widely accepted EDP concept. Elemental design patterns (EDPs) [16, 15] are just *very* small patterns that are recognized with very high reliability and very high scores.

3.3.4 Big Patterns are Seldom Classified Identically

In Section 3.3.1 and 3.3.2 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 [6] - for example, in java.io package. On the upside, however, instances of big patterns that are identically classified by more than one tool have a very high likelihood of being true positives.

4 Data Fusion Approach

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 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

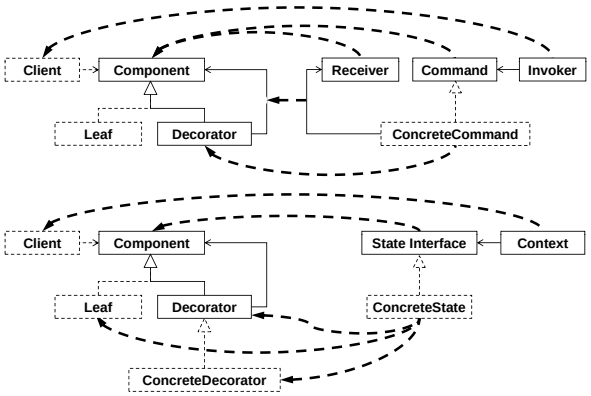

Figure 6. Witness role mappings from Command and State to Decorator

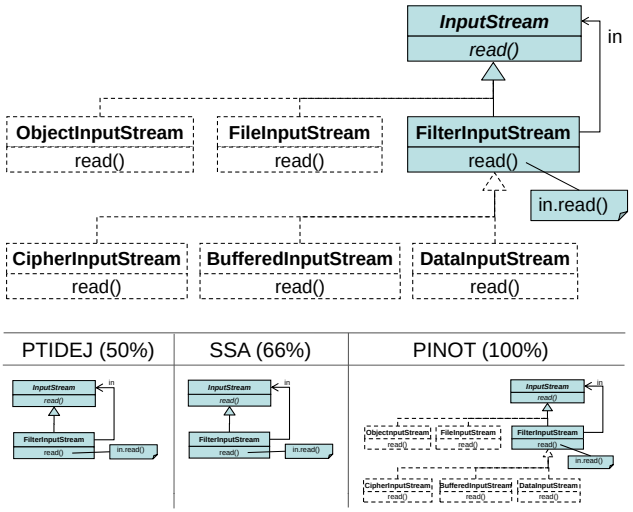

Figure 7. Three tools agree on this Decorator **instance**

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 4.2.

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 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 4.3. Witnesses are introduced first, since they are the common basis of joint recognition and reconstruction.

4.1 Witnesses

A *witness* is a pattern of which we know that some or all of its roles also play some role in another pattern. Thus any instance of a witness gives hints about the occurrence of an instance of the witnessed pattern. This is a rather general definition.We noticed several forms of witnesses depending on what "hint" means. In this paper we focus on a specific form of witnesses, leaving investigation of other forms to future work (Section 7). Leveraging on the good recognition of small patterns, we focus here on *superpatterns* as witnesses of their respective subpatterns (Section 3.3.2). For example, Command and State are witnesses for Decorator by virtue of being its superpatterns (Fig. 5).

Because superpatterns are taken as witnesses for subpatterns, the *witness role mappings* are inversed compared to the subpattern role mappings shown in Figure 4. This is illustrated in Figure 6 and 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.

4.2 Agreement and Joint Recognition

If two DPD tools A and B classify the same set of program elements as an instance of the same pattern P with more than 50% score *and* their role assignments for the mandatory roles (see Section 2) are the same, we say that *A and B agree on this pattern instance*.

Figure 7 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. 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 7) was jointly recognized by SSA, PINOT and Ptidej.

4.3 Disagreement and Reconstruction

Unfortunately, different tools agree on instances of *big* patterns quite rarely. 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

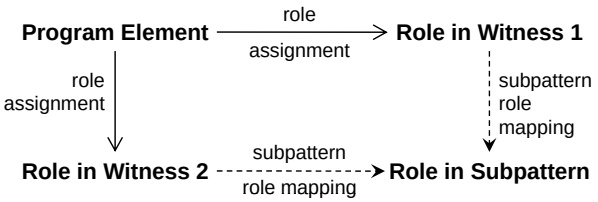

Figure 8. Consistent role mappings

agree on around 85% of Decorator witnesses. This observation motivates the following definition:

Let different tools agree with at least 66% score 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*. Two different pairs of role assignments and role mappings are *consistent* if they map the same program element to the same role, as illustrated in Figure 8.Different role mappings are *complete for P*, if they cover all the mandatory roles of P. For formal definitions see [9].

Example We demonstrate the disagreement on the Decorator instance from JHotDraw 6.0 presented in Figure 2. PINOT and DP-Miner reject this instance because 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(). SSA identifies the pattern and yields a consistent role mapping . 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 2:

- 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.

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).

4.4 Tolerating Wrongly Identified Superpatterns

Sometimes, tools fail to correctly identify a superpattern and may even agree on such wrong diagnoses. For example, for the Decorator instance from Figure 2, PINOT and SSA classify its State subpattern as a Strategy instance, since their criterion for distinguishing State and Strategy fails in this case[7]. 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.

4.5 EDPs are Witnesses too

We note that EDPs [16, 15] are witnesses too, since superpatterns are witnesses and EDPs are just very fine grained superpatterns. Thus, the witness-based approach covers the entire spectrum of granularity. This is useful since sometimes the big witnesses from Section 4.1 do not provide enough information to reconstruct a pattern. For instance, reconstruction of Observer or Composite needs an instance of the Iterator pattern as a witness[8]. Because the tools analyzed by us do not support Iterator we approximated it by the ArrayMultiReference EDP supported by

[7] In a State instance, they require 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 2.

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

		Detected Instances						True Instan -ces
		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 3. Results from the manual application of data fusion.

Fujaba. EDPs also help distinguish patterns with the same set of 'big' witnesses (see Figure 5). 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. We used EDPs also to reconstruct deviations of Visitor.

On balance, we prefer big witnesses, which are more reliable than EDPs. If tools agree on such witnesses, 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 to suitable EDPs.

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 Section 3.2, trying to detect Observer, Visitor, Decorator, Chain of Responsibility, Proxy, Bridge, Mediator and Facade instances.

Table 3 presents the results. It lists how many instances where correctly or wrongly recognized or reconstructed for each of the different patterns. For comparison, the rightmost column shows the numbers of true instances that should have been identified.

The design patterns in the first, third and fifth row are grouped together because they have the same set of witnesses (see Table 5). In these cases we fall back to the use of EDPs as additional witnesses; this is particularly useful when distinguishing between Observer and Composite. For simplicity, we take just one pattern from each group as a representative in the following.

Improved precision and recall The first three rows illustrate cases where witness-based fusion worked very well. In our experiment, we were able to recognize or reconstruct all 10 decorators, 8 of 10 Visitors and 6 of 10 Observers that we had chosen as our benchmarks. Any individual tool detected at most 7 Decorators (Ptidej), 6 Visitors (Fujaba and SSA) and 5 Observers (SSA). Thus we could clearly improve recall. 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 12% and 60% (see [9] for details). 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 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 analyzed instances of Bridge, Mediator and Facade since there were many false positives in the results of the employed tools. For instance, Fujaba, Pinot and DP-Miner return many Bridge candidates that are actually Observers. Failures occurred mainly because the analyzed tools did not recognize indirection-based variations.

Obviously, data fusion does not replace work on further improvement of existing tools. The main areas to be addressed are dataflow analysis, support for indirection and implementation of additional pattern detectors, such as the missing detector for Iterator discussed above. Name-based criteria also seem to be promising according to our evaluation. They have been pioneered by Tonella and Antoniol [17], who used variable and class names in conjunction with concept analysis in order to select the candidates which are more likely to be true positives. Data fusion will be able to take advantage of any improvements of the analyzed 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. This is not possible if tools only report roles at classe 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. Also, different tools do not assign qualitative labels (High, Low) to their scores, making them latter incomparable.

Therefore, we identified the need for common metamodel 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.

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. In the cases where the approach failed, we identified improvements that will overcome the limitations of our initial experiments reported in this paper.

6 Related Work

To the best of our knowledge, our approach is the first one that investigated an approach to design pattern detection based on data fusion[9]. However, the idea of applying data fusion in software engineering is not entirely new. For instance, Poshyvanyk et al. [13] 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. Their paper partly inspired our approach. We went one step further by combining the output of tools that already combine different techniques.

The decomposition of patterns into smaller subparts (namely EDPs) has already been suggested in SPQR [16, 15]and implemented in SPQR, Fujaba [11], EDPDetector4Java [2] and to a limited extent (one EDP) in SSA [18]. 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 [22] and could be integrated into our framework in the future.

Several approaches touch the idea of optional and mandatory roles. Liberal role checking performed in Ptidej [1] allows some "less important" roles to be missed so it is similar to our concept. Di Penta et al [12] refer 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.

Our approach is partly based on performing two analysis stages. This is similar to the approach of Lucia et al. [3].

There are certainly many parallels of our work to any other work on DPD. In this paper we have thoroughly discussed five of the currently available tools and the techniques that underlie them. For a broader discussion of other approaches we refer to the recent state of the art overview by Dong and Peng [5]. There, however, most tools were compared only based on literature, without running them. 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.

7 Conclusions and Future Work

In this paper we reported on the findings obtained by a practical evaluation of DPD tools and described how they underline a novel approach to design pattern detection based on data fusion. We introduced the concept of sub- and superpatterns and used superpatterns as *witnesses* for bigger patterns on which the evaluated tools failed. Our techniques of *joint recognition* and *reconstruction* of patterns from witnesses were shown to improve precision and recall for the analyzed instances of Decorator, Proxy, Chain of Responsibility, Visitor, Observer, and Composite beyond the level achieved by any of the analyzed 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 a common export format of DPD tools would significantly improve the results of data fusion. Next we will refine our approach based on the insights from this paper, implement it, and include emerging new tools, such as MoDeC [10], in our experiments. Furthermore, we intend to examine not only subpattern witnesses but also use other relations between patterns as indicators. For example, Mediator and Facade are structurally very similar. But Facade, unlike Mediator, is typically implemented as Singleton. So Singleton is a witness of Mediator when it is necessary to distinguish it from a Facade. Last but not least, the thresholds currently used are based on the thresholds used by some of the analyzed tools (SSA, for instance, also uses a 50% threshold). However, as we found out that scores from different tools cannot be compared, we will need a more general approach in the future.

[9] A more detailed presentation is contained in [9].

8 Acknowledgments

We are indebted to Yann-Gaël Guéhéneuc, Lothar Wendehals, Nikolaos Tsantalis and Nija Shi for providing us with their DPD tools and helping to install and use them. We also thank Yann-Gaël Guéhéneuc, Lothar Wendehals, Nikolaos Tsantalis, Clement Izurieta and Jim Bieman for many fruitful discussions. Last but not least, we thank our colleagues Daniel Speicher, Holder Mügge and Jörg Westheide for a lot of good feedback, discussions and proofreading a draft of this paper.

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

[3] 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.

[4] 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.

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

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

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

[8] 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.

[9] G. Kniesel and A. Binun. Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection. Technical report IAI-TR-2009-01, ISSN 0944-8535, CS Department III, Uni.Bonn, Germany, Jan. 2009. http://www.cs.uni-bonn.de/~gk/papers/IAI-TR-2009-01.pdf.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

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

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

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

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

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

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

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

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