High Accuracy Design Pattern Detection

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Alexander Binun
aus Kharkiv (die Ukraine)

Bonn, 2012

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn.

Erstgutachter: Prof.Armin B.Cremers, Bonn.

Zweitgutachter: Prof. Alexander Chatzigeorgiou, Thessaloniki, Griechenland.

Tag der Promotion: 15.06.2012.

Erscheinungsjahr: 2012.

Abstract

Occurrences of design patterns in existing code conveys important information to software developers about the intent of the original author. Therefore, automated design pattern detection is highly desirable when it comes to understanding unknown code. However, existing design pattern detection tools often deliver different results on the same code. Their precision and recall has so far been insufficient to make design pattern detection part of modern integrated development environments and development practices. In this context, this thesis provides several contributions to the state of art.

First, it points out that the widely disparate results are rooted in the fact that various authors model design pattern structure and behavior using different, hard to compare concepts. A realistic comparison is additionally inhibited by the lack of common test sets that could serve as a reference for the evaluation of design pattern detection tools.

Accordingly, the first contribution of my thesis is a common benchmark for design pattern detection. It consists of software repositories of varying size and complexity that include deviations from straightforward applications of design patterns, thus preventing overfitting to a particular scenario. The benchmark is the practical basis for meaningful and practice-oriented comparisons.

The conceptual basis for a comparison is provided by a formalism that allows expressing structure and behavior of pattern implementation whereas covering the concepts supported by the reviewed tools. In this formalism, design pattern implementations are viewed as sets of roles played by software entities. The specific structure and behavior of pattern implementations is captured by a set of constraints specified in a first-order-logic language. These constraints express program analyses at a generic level that can be implemented by static or dynamic analysis techniques, or any combination thereof. The current thesis explores two different instantiations of the conceptual approach:

- *Data fusion* combines the outputs of several pattern detection tools that partly use complementary techniques. The manual validation of this approach yielded better precision and recall than each of the evaluated tools [28, 42]. The first step towards the automation of this approach was the development of a common metamodel of the core concepts and results of design pattern detection tools and the specification of a common exchange format, DPDX[43, 44]. However, the data fusion approach was not implemented since it would have required implementation of DPDX in tools developed by other research groups, making the progress of my thesis dependent on their schedules. In addition, data fusion would have also combined the weaknesses of the used tools (e.g. bugs or bad run-time performance).

- *Technique fusion* combines the core ideas of various pattern detection techniques within one tool. The resulting pattern detection tool, DPJF (an acronym for **D**esign **P**attern detection by **J**oining **F**orces), achieves for all analyzed design patterns on all benchmark projects 100% precision and significant (partly many-fold) improvements of recall compared to each of the four other evaluated tools. Notably, these improvements are not bought at the expense of speed: Together with SSA [68] and PINOT [58] DPJF is the third-fast tool. The high detection quality is achieved by a well-balanced combination of existing structural and behavioural analysis techniques whereas the good performance is achieved by empirically validated simplifications of the pattern detection techniques that were combined.

By its improvement of all relevant parameters, my approach makes design pattern detection for program comprehension suitable for routine application by professional developers.

Zusammenfassung

Vorkommnisse von Entwurfsmustern in bestehendem Code vermitteln Entwicklern wichtige Informationen über die Absichten des ursprünglichen Autors. Daher ist die automatisierte Entdeckung von Entwurfsmustern ein wichtiges Hilfsmittel zum Verständnis unbekannten Codes. Existierende Verfahren und Systeme zur Erkennung von Entwurfsmustern liefern jedoch oft unterschiedliche Ergebnisse. Ihr Genauigkeit ("precision") und Entdeckungsrate ("recall") sind zu gering, um Entwufsmustererkennung zum festen Bestandteil aktueller Entwicklungsumgebungen und Entwicklungspraktiken zu machen. In diesem Zusammenhang bietet diese Arbeit mehrere Beiträge zum Stand der Technik.

Zunächst zeigt sie, dass die voneinander stark abweichenden Ergebnisse bestehender Werkzeuge auf ihrer unterschiedlichen Modelierung von Entwurfsmustern beruhen. Ein fundierter Vergleich wird zusätzlich dadurch erschwert, dass es keine Referenzprojekte zur Evaluation solcher Werkzeuge gibt.

Dementsprechend ist der erste Beitrag meiner Arbeit eine Menge von Referenzprojekten, deren sehr unterschiedliche Komplexität und Größe der Überspezialisierung von Werkzeugen auf bestimmte Szenarien entgegenwirken. Sie bietet die Grundlage für aussagekräftige und praxisrelevante Vergleiche verschiedener Ansätze und Werkzeuge.

Die formale Grundlage für fundierte Vergleiche ist ein einheitlicher Formalismus zur Spezifikation der strukturellen und dynamischen Eigenschaften von Entwurfsmuster-Implementierungen, der die Bandbreite der in den untersuchten Werkzeugen eingesetzten Techniken abdeckt. In diesem Formalismus werden Entwurfsmuster als Mengen von Rollen betrachtet die von Software-Elementen gespielt werden können. Die spezifische Struktur

und das charakteristische Verhalten von Muster-Implementierungen werden durch eine Menge von Prädikaten ("constraints") in einer auf Logik erster Ordnung basierenden Sprache dargestellt. Diese Prädikate stellen generische Programm-Analysen dar, die sich gleichermassen durch statische, dynamische oder kombinierte Analyserfahren umsetzen lassen.

Die Arbeit untersucht zwei unterschiedliche Ausprägungen diese Ansatzes:

- *Daten-Fusion* ist die Verknüpfung der Ergebnisse verschiedener bestehender Werkzeuge, die teilweise komplementäre Verfahren verwenden. Die manuelle Validierung dieses Ansatz auf den verwendeten Referenzprojekten ergab eine bessere Genauigkeit und Abdeckungsrate als die der einzelnen verwendeten Werkzeuge [28, 42]. Als wesentlicher Schritt zur Automatisierung des Fusions-Ansatzes wurde ein gemeinsames Meta-Modell zur Beschreibung der Kernkonzepte und Ergebnisse von Werkzeugen zur Entwurfsmustererkennung und ein entsprechendes einheitliches Austauschformat, DPDX, entwickelt [43, 44]. Der Ansatz wurde jedoch nicht implementiert, da die praktische Umsetzung des DPDX-Formates in Werkzeugen verschiedener anderer Forschungsgruppen den Fortschritt meiner Arbeit von externen Prioritäten und Zeitplänen abhängig gemacht hätten. Ferner hätte die Daten-Fusion auch die Schwächen (z.B. Implementierungsfehler, teilweise schlechte Laufzeit) der verwendeten Werkzeuge kombiniert.
- *Technik-Fusion* ist die Verknüpfung der Kern-Ideen bekannter Verfahren innerhalb eines Werkzeug. Das resultierende Mustererkennungswerkzeug, DPJF (ein Akronym für Design Pattern detection by Joining Forces), erreicht für alle untersuchten Entwurfsmuster auf allen Referenzprojekten 100%-ige Genauigkeit und eine deutlich (teilweise um ein vielfaches) bessere Abdeckungsrate als jedes der vier Vergleichssysteme. Bemerkenswerterweise werden diese Verbesserungen nicht durch Laufzeiteinbußen erkauft: Hinsichtlich Laufzeit erzielt DPJF (zusammen mit SSA[68] und PINOT [58]) einen beachtlichen dritten Platz. Die hohe Erkennungsqualität wird durch eine ausgewogene Kombination aus bestehenden Struktur- und Verhaltensanalysetechniken erreicht. Die sehr gute Laufzeit wird durch empirisch validierte Vereinfachungen der kombinierten Techniken erreicht.

Dank der Verbesserung aller relevanten Parameter ermöglicht mein Ansatz den bisher fehlenden routinemäßigen Einsatz von Entwurfsmustererkennung für Programm-Verstehen.

Acknowledgements

First and above all, I praise G-d for providing me this opportunity and granting me the capability to proceed successfully.

I would like to express my deep and sincere gratitude to my supervisor, Professor Armin B. Cremers from the Institute of Computer Science, Friedrich-Wilhelm-University of Bonn. His wide knowledge and the interdisciplinary way of thinking have been of great value for me. His support and understanding have provided a good basis for the present thesis.

I am deeply grateful to Dr. Günter Kniesel from the Institute of Computer Science, Friedrich-Wilhelm-University of Bonn. Günter collaborated with me throughout my thesis with his patience and knowledge, at the same time allowing me the room to work in my own way. I attribute my current scientific level to his encouragement and effort. A lot of personal support from Günter was also valuable when I relocated to Germany. So now I can say that Germany became my "scientific homeland".

I wish to express my sincere thanks to Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, Nikolaos Tsantalis, Péter Hegedüs, Lajos Jenö Fülöp and Rudolf Ferenc for long discussions on the details of their tools and approaches. From them I have learned a lot about design patterns and also how to organize personal collaborations. The approach presented in my thesis was developed and polished thanks to their feedback. This collaboration can be described by citing Isaac Newton, "If I have seen further it is by standing on the shoulders of giants".

The main point of my thesis is joining experiences of different researchers, so feedback and discussions lie at its core. Krzysztof Stencel and Patrycja Wegrzynowicz kindly consented to share valuable information relevant to several design patterns. Andrea Caracciolo and Francesca Arcelli provided me with an XML dump of the DPB database. Useful feedback and discussions were also provided by Clement Izurieta, Jim Bieman, Daniel Speicher, Holger Mügge and Jörg Westheide.

I am indebted to the JTransformer team, especially to Tobias Rho, Fabian Noth and Andreas Becker who supported me when I was implementing my pattern detection tool. Daniel Speicher and Jan Nonnen helped me use their Cultivate library in my tool.

My sincere appreciation is to DAAD (German Academic Exchange Service) for the financial support during my stay in Germany. In particular, my sincere thanks to Margret Leopold from the DAAD office in Bonn for her advice and kind support.

Last but not least, I would like to thank my family for all their love and encouragement. My beloved wife and my three children stayed in Germany with me and were always a powerful source of inspiration and energy.

Contents

Part 1. State of Art 1

Chapter 1. Introduction 2
- 1.1. Limitations of current approaches 2
- 1.2. Goals of the thesis 3
- 1.3. Contributions 4
- 1.4. Structure of the thesis 7

Chapter 2. Background 8
- 2.1. General Notions 8
- 2.2. Design Pattern Basics 10
- 2.3. Discussed Design Patterns 11
- 2.4. Pattern classification 15
- 2.5. Pattern relevant interactions 16

Chapter 3. State of Art and Problems 18
- 3.1. Benchmarks 18
- 3.2. Evaluated Tools 19
- 3.3. Evaluation Methodology 20
- 3.4. Main Problems 21
- 3.5. Conclusions 37

Part 2. Approach 1: Data Fusion 39

Chapter 4. Data Fusion Approach 40
- 4.1. Brief Overview over the Approach 40
- 4.2. Smaller patterns witness bigger patterns 41
- 4.3. Agreement and Joint Recognition 42
- 4.4. Disagreement and Reconstruction 43
- 4.5. Wrongly Identified Superpatterns 45
- 4.6. Additional Sources of Information 45
- 4.7. Evaluation of the Data Fusion Approach 46
- 4.8. Conclusions 47

Chapter 5. A Common Exchange Format for Design Pattern Detection Tools 50
- 5.1. Motivation 50
- 5.2. Requirements to an exchange format 51

5.3. DPDX Concepts 51
5.4. The architecture of a pattern candidate 53

Part 3. Approach 2: Fusion of Techniques 54

Chapter 6. An Object Level View of Design Patterns 55
6.1. Pattern occurrences at the object level 56
6.2. The Core Constraints 56
6.3. The object level vs. the program element level 60
6.4. Conclusion 63

Chapter 7. Specifying Pattern-Specific Interactions 64
7.1. The Software System Abstraction 64
7.2. Roles and Relationships 71
7.3. The Constraints 74
7.4. Behavioral Pattern Specifications 86
7.5. Summary 91

Chapter 8. Implementation Issues 94
8.1. Implementing the Pattern Interaction Framework 95
8.2. Improving Recall 98
8.3. Improving Response Time 99
8.4. Combining of static and dynamic analyses in constraint implementations 104
8.5. The organization of DPJF 105

Chapter 9. Evaluation of DPJF 111
9.1. Computing precision and recall 111
9.2. Discussion of Results for Precision and Recall 114

Part 4. Related Work and Conclusions 124

Chapter 10. Related Work 125
10.1. Efficient Implementations of Behavioral Analyses 125
10.2. Design Pattern Detection Approaches 125
10.3. Specifying interactions between pattern participants 128
10.4. Benchmarking and Tool Comparison 128
10.5. Compiling Test Sets 130
10.6. Data Fusion 131

Chapter 11. Conclusions and Future Work 132

Bibliography 135

Appendix A. Structurally similar patterns 140

Appendix B. Response Time Statistics 143

Part 1

State of Art

CHAPTER 1

Introduction

Design patterns [27] (referred to in the thesis as *patterns*) are domain-independent descriptions of groups of communicating classes that present solutions to recurring object-oriented design problems. Uncovering patterns from source code is therefore important for program comprehension. A pattern detection tool returns a list of *candidates*. In order to evaluate the accuracy of the tool, the returned candidates are compared against known pattern *occurrences* in the analyzed program.

1.1. Limitations of current approaches

Pattern detection is complicated by the high number of structurally and behaviorally diverse *implementation variants* that emerge when a programmer adapts informal descriptions of the collaborating entities to a concrete context. The structure and behavior of pattern implementation variants are represented differently in various approaches (see [8, 12, 15, 20, 71, 33, 34, 70, 36, 67, 72, 17, 26] for theoretical approaches and [2, 16, 52, 68, 58] for implemented tools). Analyzing these approaches, the following limitations that motivated my research were identified:

Inconsistency of pattern detection results: Existing pattern detection tools often render different, contradictory results on the same code.

Low precision: Precision is the ratio of pattern candidates detected by a tool to the number of correct pattern occurrences among the suggested candidates [17]. Ideal precision (100%) would correspond to a tool that is fully trustworthy in that each suggested candidate is a real pattern occurrence. Existing tools often yield a low precision, with the values for individual benchmarks ranging, according to the measurements provided in Chapter 9, from 9% for PINOT [58] to 66% for Similarity Scoring [68].

Low recall: Recall is the ratio of pattern candidates detected by a tool to the number of correct pattern instances within the test set [17]. Ideal recall (100%) would correspond to a tool that detects all possible implementation variants of each supported pattern. Existing tools often yield a low recall, with the values for certain benchmarks ranging , according to the measurements provided in Chapter 9, from 18% for Ptidej [29] to 44% for SSA [68].

Partly low speed: Some existing tools respond very slowly to user queries and run fast on some programs whereas being slow on others. According to the measurements provided in Section 9.2, Ptidej [29] detects Observer and Composite patterns in JHotDraw 6.0 in 69.9 sec. (40 times faster than the fastest tool). In JavaIO, Ptidej detects Observer and Composite patterns in only 2.3 seconds (7 times faster than the fastest tool).

The net effect of all these problems is that existing approaches are interesting research contributions but none has a significant community of regular professional users. Obviously, all known tools and approaches are still far from being perceived as useful by professional developers. Therefore, the aim of this thesis is the development of a methodologically, empirically and formally well-founded approach that provides the basis for a pattern detection tool that is ready for professional use.

1.2. Goals of the thesis

The goal of my pattern detection approach is to ensure the following quality characteristics:

Ideal precision: Precision close to 100% is the key prerequisite for letting developers trust any analysis tool.

High speed: Analyses must be perceived as "fast" by their users. This is clearly a subjective measure that depends on the concrete use case. When a pattern detection tool is used as a helper in a one-time program comprehension task, developers might be willing to wait tenths of seconds (or even minutes on very large programs) because the tool's result will still appear much faster than any manual search could do. When a pattern detection tool is used as a development aid that hints to possible problems in the application of design patterns, its output must be as soon as possible available after saving changes to some file. This corresponds to delays of at most a few tenths of a second. The aim is to push speed to the limit where such novel applications of pattern detection as permanently active development aids become possible.

Good recall: To be ready for everyday use, recall of automated pattern detection must be significantly improved compared to the one of existing tools. A tool must be able to guarantee detection of all commonly used implementation variants of a pattern. However, it may fail on exotic variations whose automated treatment would seldom be needed but would adversely impact precision or run-time or both.

Design pattern detection, like any information retrieval task, suffers from *false negatives* (missing correct occurrences) and *false positives* (incorrect classification of pattern candidates) - see [17]. These are conflicting issues.

Pattern detection tools that apply liberal detection criteria accept more candidates. On the flip side, the number of false candidates might also grow. In contrast, tools that apply too restrictive detection criteria filter out many false positives, but correct occurrences may be filtered out as well. In addition, the need for speed adversely affects the number of false negatives and positives. The different techniques to achieve precision, recall or speed negatively influence each other. By the start of this thesis it was commonly accepted in the design pattern community that it would be impossible to improve in one discipline without degrading in another. This thesis contradicts this "old wisdom", showing that it is even possible to significantly improve in two disciplines without degrading in the third.

1.3. Contributions

Evaluation of pattern detection tools on a common benchmark. After examining various pattern detection approaches, it was figured out that the test sets used by various experts to develop and evaluate their tools are very diverse. Test sets usually represent the most frequently encountered implementations (or simply those found manually by an expert). These are mostly straightforward ones. The more non-trivial implementations are taken, the more disagreement between various experts emerge and the more diverse test sets are. This problem is referred to in this thesis as the *biased test set* problem since a test set reflects the experience of an individual expert, i.e. biased towards her or his perception. The biased test set problem is aggravated by the lack of documentation, which leaves room for speculation about the intent of the author of a particular piece of code, inviting divergent interpretations by others (as noted by Pettersson et al. [53] and Dong et al. [17]).

To overcome the biased test set problem, an extensive practical evaluation of different pattern detection tools that employ different static and dynamic analysis techniqueswas performed. The evaluation results were summarized in Chapter 3. The reviewed pattern detection tools were evaluated on various code repositories of various sizes and complexities that are described in Sec. 3.1. The test sets were prepared by screening these repositories manually and by running the tools evaluated in Chapter 3. These activities were done in collaboration with the authors of these tools. Each potential true and false positive was thoroughly discussed, thus combining the expertise of several research groups in this thesis.

A formal framework for specifying design patterns. While evaluating tools and communicating with their authors, it was concluded that the research community needs a common formal basis that allows researchers to decide why a particular candidate is indeed a true occurrence.

For the patterns addressed in this thesis, different implementation variants of each individual pattern were reviewed and a formal framework (further referred to as *the Pattern Interactions Framework*) to express interactions in pattern occurrences was compiled. This framework is introduced in Chapter 7. Pattern interactions is formally presented as a set of roles played by software elements and relationships between roles that reflect collaborations (structural and behavioral) between role players. The *constraints* restrict the possible role players, making them collaborate in a pattern-specific way. The constraints are expressed using first-order-logic. By using different ways to connect roles with relationships and constraints one can capture a (possibly) infinite set of pattern implementation variants.

The key point in which my approach differs from prior work on pattern detection is that it captures the structural and behavioral characteristics of design patterns in the most generic terms adopted from [27], i.e. in terms of collaborations between *objects*. It was decided to depart from the commonly adopted specification of patterns at the *program element* level since descriptions at object level are believed to be better suited to capture the behavioral aspects (e.g. control- and data flow) that are essential for the interactions between pattern entities. In contrast, premature focus on a description at program element level bears the risk of obscuring that different program structures essentially implement the same object interaction. Object-level specifications make it easy to identify common characteristics obscured otherwise by mere implementation choices. Object-level characteristics are later translated into their program-level counterparts, making it explicit that one core object interaction can be implemented in several ways.

A second but equally important benefit of starting from object interactions is that the formal basis also applies to approaches based on dynamic analysis. At run-time, objects and sequences of object actions / events are the only observable concepts, whereas program structures do not exist anymore. Thus a common formal framework for static and dynamic analyses is built. The explicit mapping of object-level to program element level descriptions allows to compare dynamic and static approaches that are otherwise described independently at the two different levels.

In summary, design pattern experts can focus first on the concepts that are essential for a particular pattern and then, in the implementation phase, to find the most suitable combination of static and–or dynamic detection techniques for the specified concepts.

Fusion of outputs. Inspired by the idea that various detection techniques (for example, static vs. dynamic behavioral analyses) can be beneficially combined (as suggested by Poshyvanyk et al. [56]), it was shown that such a combination provides advantages already at a very coarse-grained level, where the *outputs of existing pattern detection tools* are combined. The related *data fusion approach* for design pattern detection is introduced in Chapter 4. It is motivated by the consideration that static and dynamic program analyses can be combined. On the one hand, static analyses make every property true of the model mapped to a true property of the program; on the other hand, they are imprecise. Static analyses are compensated by precise dynamic analyses that, however, do not guarantee that every property true of the program is true of the model (program traces). Each tool, analyzing certain aspects of a pattern, contributes its estimates, without requiring any expensive reimplementation of already available techniques.

The fusion-based pattern detection approach was manually evaluated, showing that it improves precision and recall in comparison to individual pattern detection tools whose outputs were fused.

A common metamodel and exchange format for pattern detection tools. An obstacle to the implementation of automatic result data fusion was the diversity of the output formats of the existing tools. This is at a first glance a purely syntactical issue. However, it hides a much more fundamental semantic issue: The different, often incompatible definitions of some very basic concepts on which the various tools were built - for example, how to group program elements into pattern detection candidates. Addressing this limitation, a common exchange format for DPD tools, DPDX, was developed and described in Chapter 5. DPDX is based on a well-defined and extensible metamodel resolving a number of limitations of current tools. The employed XML-based metamodel can be easily adopted by existing and future tools providing the ground for improving precision and recall when combining their findings. In the process of building DPDX, some central notions of design patterns that lacked common, generally accepted definitions were clarified. Thus a sound common foundation and terminology for pattern detection issues were provided.

A pattern detection tool. The observations reported in Chapter 3 showed that some pattern detection tools implement time-consuming program analyses that do not always contribute to the accuracy - for example, transitive closures of the object flow relation (Stencel et al. [64]). More importantly, when building on existing tools, certain combinations of techniques cannot be achieved at all or only at the price of including also some unwanted techniques.

Motivated by these observations, it was decided to depart from realizing the Pattern Interactions Framework (mentioned in Section 1.3) through the data fusion. Instead, several pattern detection techniques were combined by implementing them in one tool via static analyses.[1] More precisely, *only certain variants of these techniques* are combined because of efficiency considerations. For example, the transitive closure of the data flow relation is not implemented.

[1]According to the information combination taxonomy of Croft et al. [14] I fused algorithms instead of fusing outputs.

A pattern detection tool (named DPJF, 'Detect Patterns by Joining Forces') was implemented. DPJF yields a drastic improvement over existing tools regarding the achieved precision and recall. The response time of DPJF competes with the response time of the fastest existing pattern detection tools reviewed in this thesis. The corresponding evaluation results are presented in Chapter 9.

In summary, a pattern detection tool presented in this thesis removes several key obstacles that up to now prevented professional developers from using pattern detection tools: bad precision, recall and speed. The availability of the DPJF prototype (see http://sewiki.iai.uni-bonn.de/research/dpd/dpjf/start) enables empirical validations of the motivation behind pattern detection. Other factors that might also prevent adoption of pattern detection (e.g. questioning of patterns in general) can be also be separately investigated.

1.4. Structure of the thesis

Chapter 2 introduces the terminology used throughout the thesis. Chapter 3 presents state of art, by evaluating five existing pattern detection tools and stating the problem tackled in the thesis. Chapter 4 presents an instantiation of the Pattern Interactions Framework by fusing the outputs of different pattern detection tools. Chapter 5 presents a common exchange format for pattern detection tools. Chapter 6 informally discusses how the structure and the behavior of some pattern implementation variants can be expressed in the terms of collaborating *objects*. Chapter 7 presents the Pattern Interactions Framework formally. Chapter 8 presents DPJF, the novel pattern detection tool that instantiates the Pattern Interactions Framework by combining several static program analyses. Chapter 9 evaluates DPJF and compares the results of DPJF and of several other tools. Chapter 10 surveys related work. Finally, Chapter 11 concludes and outlines future work.

CHAPTER 2

Background

2.1. General Notions

Analyzed languages. This thesis addresses design patterns for class-based, statically typed languages (for example, Java , C++, C#) since they possess features like inheritance and static typing exploited in many pattern implementations[27]. Due to the focus of the thesis, the term “object-oriented” is used as an equivalent to “class-based, statically typed”.

Information retrieval. Assume that an information retrieval tool T detects some concept in a data set (e.g. a design pattern in the source code). T reports manifestations of this concept referred to in the thesis as *candidates*. Candidates are compared against *occurrences*, i.e. actual manifestations of the sought concept in a data set confirmed by the author and–or by an expert. For example, a design pattern detection tool searches for the implementations of a certain pattern in the source code, retrieving pattern candidates. These are compared against the occurrences in the corresponding test set.

Denote the set of candidates that are occurrences (*true positives*) by TP, the set of candidates that are no occurrences (*false positives*) by FP, the set of occurrences that are no candidates (*false negatives*) by FN and the set of remaining entities that are neither occurrences nor candidates as true negatives (TN). The *accuracy* of retrieved information denotes the fraction of true results among all pattern entities. Formally,

$Accuracy = \mid TP \mid + \mid TN \mid / (\mid TP \mid + \mid FP \mid + \mid FN \mid + \mid TN \mid)$.

Two main metrics are commonly used to reason about the accuracy (see e.g. [14]):

- *Precision* is the fraction of true positives among all candidates. Formally, $Precision = \mid TP \mid / (\mid TP \mid + \mid FP \mid)$.
- *Recall* is the fraction of true positives among all occurrences. Formally, $Recall = \mid TP \mid / (\mid TP \mid + \mid FN \mid)$.

It is also common practice in information retrieval to use integrations of these two metrics, for example, the harmonic mean of precision and recall (F-Score) : $FScore = 2/(1/Precision + 1/Recall)$.

Fmentioned by Croft et al. [14].

Note that some information retrieval tools rank their results, expressing their confidence that a particular candidate is actually a correct occurrence

		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

FIGURE 2.1.1. Program analyses

by a score. Scores are typically expressed by a percentage, with 100% indicating total confidence. Scores might be taken into account when computing precision and recall of pattern detection tools as Pettersson et al. [54] reports. This thesis presents a novel approach to compute precision and recall of a pattern detection tool that reports scores; see Chapter 9 for more details.

Program Analyses. Most pattern detection algorithms match descriptions of the expected structure and the behaviour of sought patterns against the information about the structure and behaviour of program elements. Relevant program analyses can be categorized in two orthogonal dimensions, with two subcategories each:

- **What to analyze**. *Structural analyses* deal with the structure of program entities (for example: subclassing, method signatures, variables types). *Behavioral analyses* deal with program behaviour (control/data flow).
- **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.

Figure 2.1.1 illustrates this classification.

Since behavioral aspects constitute the most essential part of pattern specifications, insufficient behavioral analyses yield extra false positives (Antoniol et al. [3], Fülöp et al. [25]).

Dynamic behavioral analyses (advocated by Gueheneuc et al. [29], Detten et al. [71] and De Lucia et al. [47]) can be effective, in particular when applied in addition to structural criteria because they are more precise. But dynamic analyses do not guarantee that every property true of the program is true of the abstraction (i.e. program trace(s)) because of hardly achievable test coverage. In contrast, static analyses are applied because they guarantee that every property true of the abstraction mapped to a true property of the program. However, static analyses are imprecise. In this thesis, static and dynamic analyses are referred to as *complementary* techniques and can be combined (Chapter 4).

Increasing the number of execution traces might lead to a loss of efficiency and covering all traces is practically impossible on large programs since it is equivalent to the problem of path coverage (Nielson et al. [51]) which is well-known to be exponential. But full-scaled static behavioral analyses might be also inefficient (Nielson et al. [51]). Tackling this issue, Sridharan et al. [63] propose an efficient *demand-driven* points-to analysis algorithm.

2.2. Design Pattern Basics

The following is an informal summary of basic concepts of design pattens and design pattern detection extracted from literature [2, 3, 23, 6, 7, 8, 9, 10, 11, 12, 47, 16, 64, 20, 21, 27, 34, 37, 39, 40, 28, 43, 46, 19, 52, 54, 57, 58, 36, 73]. They are treated formally in Chapter 7.

Definition. According to Gamma et al. [27], "design patterns are descriptions of communicating entities that are customized to solve a general design problem in a particular context". A pattern description is divided into four parts:

- *Name* - identifies the pattern so that it can be discussed.
- *Problem* - when to apply the pattern . It can be divided into intent, motivation, and applicability.
- *The description of interactions* between entities that form a pattern occurrence (also referred to as a *solution* in [27]).
  - At the program element level, *implementation variant(s)* incorporate this description and result from adapting (informal) pattern descriptions to specific situations. Several implementation variants might be abstracted into a more high-level description referred to as *a motif* in [30, 2, 31, 29, 33, 1]. Each pattern description in [27] is followed by examples of implementation variants for this pattern.
  - At the object level, interaction descriptions are expressed in the terms of memory location references and objects sending messages each to other.
- *Consequences*: description of the software after applying the pattern.

Roles, Relationships. Assume that P is a currently addressed pattern. It can be viewed as a set of *roles* played by software entities (objects, their types, methods, fields, method calls). Roles are connected by *relationships* (e.g. containment, control flow) representing interactions between their players.

An occurrence of P can be seen as a set of software entities $\{e_1 ... e_n\}$ that play some roles of P and exhibit the relations that are expected to hold between the players of these roles. A pattern detection tool basically tries

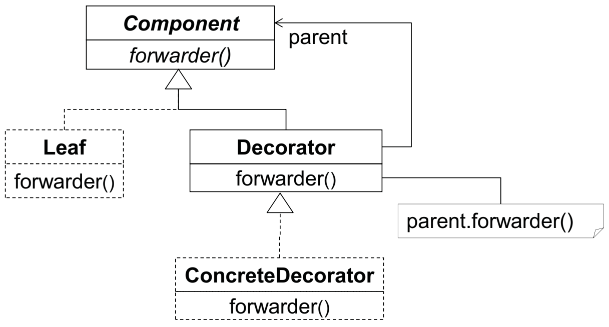


FIGURE 2.2.1. The Decorator pattern structure. Mandatory (respectively, optional) roles are drawn using solid (respectively, dashed) lines.

to identify entities that fulfill these conditions. To express its supposition it assigns entities to roles. A pattern *candidate* is a set of role assignments reported by a pattern detection tool.

Roles can be mandatory or optional. Each of the *mandatory* roles of P must be played by at least one program element in an occurrence of P. Roles whose players may be missing in some occurrence are called *optional*. The role assignments are *complete* for P if they cover all the mandatory roles of P.

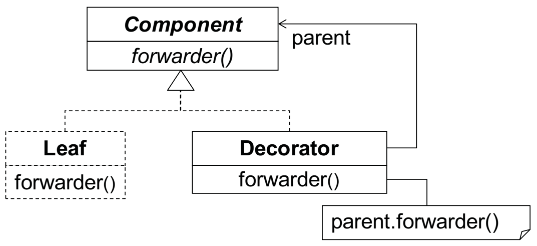
The UML diagram in Fig. 2.2.1 illustrates the structure of the Decorator pattern, displaying each of the roles named and spelled exactly as they are in the text.

- Mandatory roles: The mandatory roles cover the concepts of object chaining and request forwarding. The class playing the DECORATOR role owns the method(s) playing the FORWARDER role that forward(s) decorating requests. Forwarding calls are described by statements $parent.forwarder()$. The field *parent* plays the PARENT role and references the object to which decorating requests are addressed.
- Optional roles: Forwarding continues unless an object playing the LEAF role is reached. Also, forwarder methods can be overriden in the classes playing the CONCRETE DECORATOR role.

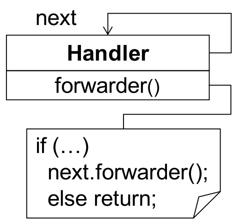
Throughout the thesis, small capital letters highlights role names throughout the thesis. This convention enables concisely saying "The DECORATOR field *parent* “ or “*parent* is the PARENT FIELD” instead of "The field *parent* that plays the DECORATOR role".

2.3. Discussed Design Patterns

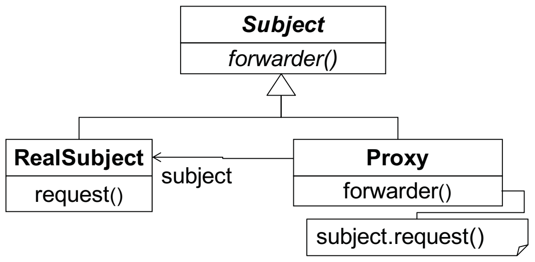
This section summarizes the intentions behind certain collaborational patterns and the main ideas behind the interactions in the occurrences of



(A) Decorator



(B) CoR



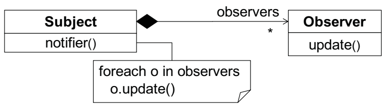
(C) Proxy

FIGURE 2.3.1. Decorator, CoR and Proxy

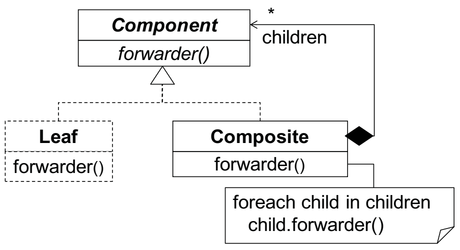
these patterns. Patterns are grouped according to structural similarity. Pattern descriptions in this section are based on

- The discussions provided in the relevant literature, e.g. in [27, 18, 57].
- Empirically validated heuristics following thorough examination of tens of occurrences taken from the analyzed benchmarks and discussed with the authors of the reviewed tools. Benchmarks and tools are provided in Chapter 3.

Decorator, Chain of Responsibility, Proxy. All three patterns deal with "chains" of objects, describing how an action is passed along the chain. The calling and called methods have the same signature in order for an action to be defined and processed uniformly. The patterns are illustrated on the UML diagrams in Fig. 2.3.1.



(A) Observer



(B) Composite

FIGURE 2.3.2. Observer and Composite

Decorator. The Decorator pattern attaches any number of additional responsibilities (encapsulated in DECORATOR objects) to LEAF objects at runtime. DECORATOR objects are linked together into "forwarding chains" using a PARENT field that can store either a DECORATOR or a LEAF object. Therefore chains of any length can exist.

The FORWARDER method of a decorator object calls the FORWARDER method of the (single) "parent" object at least once in any execution. This is done by issuing a FORWARD call(s).

Chain of Responsibility (CoR). The CoR pattern arranges the collaboration between HANDLER objects such that each one performs some part of a common task. Handlers are linked in 'forwarding chains' through a NEXT field that keeps the HANDLER object execution to be passed to. Like in Decorator, chains of any lengths can exist.

The FORWARDER method of a HANDLER object does not call the FORWARDER method of the (single) next handler object in each program execution; in some execution it just returns a result when a task is completed or performs some local actions (without forwarding to the next object).

Proxy. The Proxy pattern links a PROXY object to another (REAL SUBJECT) object so that the proxy identity can be used instead of the one of REAL SUBJECT. To facilitate forwarding, a proxy object keeps the Real Subject in a SUBJECT field. Therefore "proxy chain" lengths are usually limited (at most 2 proxy objects are allowed).

The FORWARDER method of a proxy calls the REQUEST method of a REAL SUBJECT object when appropriate (no constraints are imposed). This is done by issuing a FORWARD call(s). Usually the REQUEST method and the corresponding FORWARDER method have the same signatures.

Observer, Composite. Both patterns involve an object structure consisting of the "main object" and a set of objects related to the main object. An action, being initiated at the main object, is propagated to *every* related object. These patterns are illustrated by the UML diagrams in Fig. 2.3.2.

Observer. The Observer pattern keeps the system state consistent by propagating state changes occurring at a SUBJECT object to several OBSERVER objects whose states conceptually depend on the subject's state. A SUBJECT object is responsible for managing the fields(s) storing OBSERVERS. The values of these fields should be modified after a SUBJECT is constructed, so that individual OBSERVERS objects can be added or deleted at any time. To propagate state changes, a NOTIFIER method of a SUBJECT object triggers reactions to state changes in its observers by calling the related UPDATE method on each OBSERVER object in each own execution. This is done by issuing a UPDATE call(s).

In order to preserve a consistent system state all subject state changes must be completed before notifying the dependent observers.

Composite. The aim of the Composite pattern is to perform operations defined on tree-like, recursive object structures. Some nodes serve as the entries, LEAF objects represent primitive objects and other nodes represent CONTAINER objects. A CONTAINER object owns the CHILDREN fields(s) storing child objects. The values of these fields might be modified after a CONTAINER object is constructed, so that children objects can be added to/deleted at any moment.[1]

Different FORWARDER methods are defined for each tree node but all have the same signature. A FORWARDER method issues FORWARDING calls. Thus, a call of the same operation is forwarded to each subnode of the container object.

Visitor. The intent of the Visitor pattern is to implement an OPERATION method outside of the OBJECT STRUCTURE on which it operates and without tightly coupling the OBJECT STRUCTURE to the OPERATION that is applied on ELEMENTS of this structure. This operation is represented by the method(s) of a visitor object. An ELEMENT object owns ACCEPT methods that facilitate traversals by calling ACCEPT methods on conceptually related elements or calling VISIT method(s) of a VISITOR object passed as a parameter, or both. In VISIT METHOD CALLS the reference to the current structure element is passed as a parameter so the proper VISIT METHOD is finally executed.[2] The double-dispatch technique is crucial in implementing these interactions.

[1]Post-construction update of observers / contained objects is not a strict requirement since there are rare Observer / Composite occurrences where the set of observers / contained objects remains unchanged after construction. For example, the class QuadTree in JHotDraw 6.0 represents a composite object that owns its parts (north, south, east and west). These parts are initialized in the QuadTree constructor and henceforth remain unchanged.

[2]A visit method might, in turn, invoke accept method(s) on related elements, thus continuing a traversal.

2.4. Pattern classification

The pattern classification in this thesis slightly differs from the classification of Gamma et al. [27]. The reason behind the regrouping done in this thesis is that the classification of Gamma et al. [27] deals with the purpose and is useful for programmers, but is not accurate in the reverse-engineering sense. For example, the structural part of the Composite pattern (a structural pattern) requires a container C to keep a collection of the "children" object and a forwarder method F owned by C to invoke the method M with the same signature on the children. In the Composite pattern, there is a behavioral requirement that was established by me by analyzing tens of true and false Composite candidates: F must invoke M on each child object in every own execution. The presence of behavioral aspects in the Singleton pattern and the need to regroup patterns for reverse-engineering activities was also mentioned in [58].

This thesis groups patterns "in the reverse-engineering sense" by the kind of interactions between the software entities. The patterns are divided into *collaborational patterns* (Sec. 2.4) and *creational patterns* (Sec. 2.4). The interactions in the collaborational patterns are driven solely by structural and–or behavioral collaborations. The interactions in the creational patterns are driven, in addition, by object creations.

Collaborational patterns. Most of the patterns addressed in this thesis (e.g. Decorator, Proxy, Chain of Responsibility, Observer, Composite, Visitor, Adapter, Command, State, Strategy) focus solely on inter-object collaborations and belong therefore to the collaborational group. In a collaborational pattern, an object O referred to as *the master object* owns *the caller methods* $M_1 ... M_k$ that invoke *the callee methods* $m_1 ... m_c$ on the object(s) $o_1 ... o_m$ conceptually dependent on O. These objects are pointed to by *the dependent object field(s)* $F_1 ... F_d$ owned by the master objects. For example, in the Observer pattern the master object is a subject that owns the notifier methods (that stand for the caller methods). Update method(s) stand for the callee methods and are invoked on the observers (which stand for the dependent objects).

If the signatures of a caller method M and a callee method m are identical, it is said that M *forwards* to m. For example, in the Proxy pattern the request method owned by a proxy forwards to the concrete operation method owned by the real subject.

Decoupling. Certain collaborational design patterns (e.g. Observer, Composite, State, Strategy, Bridge) facilitate decoupling of master objects from their dependent objects. Such patterns are referred to in this thesis as *decouplers*. For example, in the Observer pattern the contents of the set of observer objects associated with a given subject S might change after S is constructed. In the Strategy pattern, the context object C might initiate a change in the contents of the field pointing to the strategy object after C is constructed.

Creational patterns. The interactions in Abstract Factory, Factory Method, Singleton, Prototype and Builder involve object creations. These patterns comprise the creational group. In a creational pattern, a *creator method* owned by the corresponding *creator object* creates *product object(s)*. For example, in the Abstract Factory pattern the creator methods are owned by the "concrete factory" objects and create object of the "concrete product" type.

2.5. Pattern relevant interactions

This section provides the terminology to discuss pattern-relevant interactions. The names of the entities of the object-oriented paradigm (fields, variables, methods, types) are consistent with the terminology of Craig [13].

Structural Interactions.

- Ownership:
  - An object O owns *instance variables (or fields)* and *methods*. A type T owns *static (or shared)* fields. The standard denotation $O.f, O.M$ for a field f (or an instance method M) of O is used. A shared variable v of a type T is denoted by $T.v$
  - A method M might have *local variables* $v_1 ... v_n$; these are denoted as $M.v_1 ... M.v_n$.
  - Pattern-relevant fields: For fields $O.f_1 ... O.f_n$, there is a subset $\{f_1 ... f_k\} \subseteq \{f_1 ... f_n\}$ of fields called *the relevant fields* in the context of some design pattern(s). A relevant field $O.f$ points to some objects $o_1 ... o_m$ conceptually related to O. It might happen that $m > k$ since some fields might point to more than one object at a time.
- Entity types:
  - The subtyping relation is denoted in this thesis as $<$. Writing $T_1 \leq T_2$ means that T_1 is a subtype of T_2. Writing $T_1 < T_2$ means that T_1 is a strict subtype of T_2.
  - The static type of a variable v (at the program element level) is denoted by $type(v)$; the type of an object o (determined by its class) is denoted by $type(o)$.
  - The types t_1 and t_2 are *sibling types* if and only if they are not involved in the subtyping relation and have a strict common supertype, i.e. $t_1 \nleq t_2 \wedge t_2 \nleq t_1 \wedge \exists t : t_1 < t \wedge t_2 < t$.

Behavioral Interactions.

Control flow. A certain method M launches a method call mc *directly* when the statement corresponding to mc resides within the body of M; otherwise it does this *indirectly*. In case of an indirect call *intermediate method calls* are launched before mc is reached. Intermediate method calls refer to *intermediate methods* which are finished after mc is executed. In Figure 2.5.1 intermediate methods are $I_1 ... I_n$.

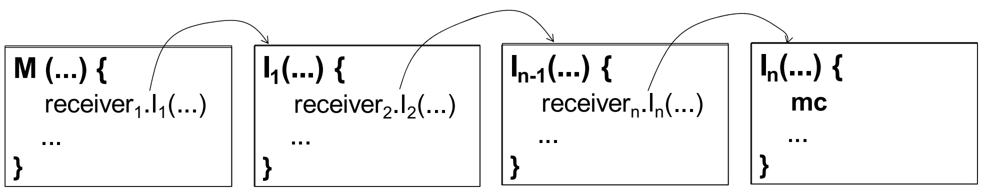


FIGURE 2.5.1. Indirect invocation

If M invokes mc directly or indirectly, mc is said to be *control flow reachable* from M. Alternatively, mc is said to be *in the control flow* of M.

Discard control flow paths that do not reflect pattern logic. Certain program executions do not represent application logic typically reflected by patterns. They are therefore excluded from analysis when control flow reachability is checked or individual executions (in Partial / Full Coverage constraints, Sec. 7.3) are examined.

According to the author's observations, the following paths in the control flow graph do not reflect pattern logic:

- A path taken when an exception is processed. Such a path usually contains cleanup actions done in an abnormal scenario. These actions usually reside in "catch" or "finally" blocks and represent closing file handles, network connections etc.
- A path through a loop that consists only of the continuation test and its failure branch. Such a path does not carry useful information. Only paths that actually enter the loop might contain conceptually relevant method calls are therefore are of interest.
- A path corresponds to two or more cycles in a loop. Only individual cycles are examined in order to ensure that certain invocations are performed on certain object(s) o in *each* possible control flow path through the loop.

The control flow paths that do not fall into the above mentioned categories are said to be *pattern-relevant.*

Data flow. If objects are copied from a variable u to a variable v due to certain actions corresponding to variable assignments, method parameter passing or returning of method results, *the dataflow from u to v* occurs; v is said to be *dataflow reachable* from u. If the actions causing this dataflow are in the control flow of a method M , *M is the scope of this dataflow.* When a variable v points to objects $o_1...o_n$ it is written $pointsTo(v) = \{o_1...o_n\}$.

In user-defined libraries the object that marks the start of a dataflow chain might be absent, instead residing in a client program that initiates dataflow. It is assumed that such "incomplete dataflow" starts at an *anonymous object* of the corresponding type.

CHAPTER 3

State of Art and Problems

This chapter describes a practical evaluation of five state-of-the-art pattern detection tools on the benchmarks from Section 3.1 and outlines the main problems that are tackled in this thesis. Section 3.2 describes the evaluated tools. Section 3.3 presents the evaluation methodology. The evaluation findings discussed in Section 3.4 constitute the problems to be addressed in this thesis. Finally, Section 3.5 provides the main conclusions.

In all source code examples throughout this section, data flow is depicted by thick pointed lines; control flow is depicted by thick dashed lines. Source code is accompanied with role assignment information. A role is illustrated by an oval box with the role name written within. The thin solid arrow(s) from a box representing a role R lead(s) to the program element(s) playing R. A Decorator occurrence in Figure 3.4.2a illustrates these denotations. In false candidates, role names are followed by question signs; see an example of a false Decorator candidate in Figure 3.4.2b.

3.1. Benchmarks

Reviewing the relevant literature and running publicly available pattern detection tools, it was observed that these might perform well on the well-known projects commonly used as reference but poorly otherwise.

Therefore in order to perform a trustworthy evaluation it was decided to take code repositories of various sizes and complexities. See Fig. 3.1.1 for the repository sizes. The tool evaluation was carried out on six projects that cover a wide range of pattern-relevant scenarios:

- **Known samples:** JHotDraw 5.1 and JHotDraw 6.0 are small libraries designed for teaching purposes, which contain many straightforward to detect pattern occurrences that closely follow the descriptions by Gamma et al. [27]. These are included because their actual pattern occurrences are well-known.
- **Standard libraries:** Java IO 1.4 and Java AWT 1.3 are example of widely-used, industrially developed libraries that are indispensable in any non-trivial application. Their particular challenge to pattern detection tools are apparently incomplete pattern occurrences, where some role players are completed by the applications using the libraries.
- **Large applications:** ArgoUML 0.18.0 is a large application implementing an open source CASE tool. Being self-contained it does

not impose the incompleteness problem of libraries. However, its size is a challenge for many tools.

- **Applications with a lot of implementation variants:** TeamCore is a component of the Eclipse 3.6 rich client platform, a large, quickly evolving industrial application containing very unusual DP implementation variants.

Figure 3.1.1 presents these repositories. Note that this benchmark can only make an evaluation more trustworthy without claiming that it is optimal or even that an optimal set of benchmarks exists.

	JHotDraw 5.1	JHotDraw 6.0.1	Java IO 1.4	AWT 1.3	ArgoUML 0.18.0	Eclipse 3.6 TeamCore
packages	64	84	30	76	200	41
classes	1.002	1.732	517	1.653	3.776	702
fields	3.188	4.518	1.062	6.123	8.518	1.210
methods	8.793	14.975	4.471	14.618	31.577	5.514

FIGURE 3.1.1. Size of analysed projects including their source code *and* all the referenced bytecode.

3.2. Evaluated Tools

The tools were selected so as to support as many basic pattern detection approaches as possible. From the 14 pattern detection approaches for Java listed in the state of the art overview of Dong et.al. [17], only 8 were found to be available for practical experimentation[1]. In addition, the tools that detect as many pattern implementation variants as possible were favoured. Based on this criteria five tools listed in Table 1 were selected. They are:

- SSA (Similarity Scoring) [68] performs static analyses of inter-class relationships and method signatures. Call graphs are built using the coarse-grained algorithm CHA (Tip et al. [66]). Limited transitivity of the inheritance relation is supported.
- DP-Miner [16] performs thorough static behavioral analyses. No dynamic analyses are performed.
- PINOT [58] performs thorough static behavioral analyses when detecting certain patterns, which can sometimes be even a bit too restrictive. No dynamic analyses are performed.

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

	Static structural	Static behavioral	Dynamic structural	Dynamic behavioral
SSA	Associations, inheritance, signatures	No	No	No
DP-Miner	Associations, inheritance, signatures, names	Control flow and data flow	No	No
PINOT	Associations, inheritance, signatures	Control flow and data flow	No	No
PTIDEJ	Associations, inheritance, signatures	Object creation	Association versus aggregation	Control flow and data flow (partial)
FUJABA	Associations, inheritance, signatures	No	Association versus aggregation	State versus Strategy

Table 1. Analysed tools and the basic techniques that they apply.

- Ptidej [31] treats pattern recognition as a constraint satisfaction problem. Implementation variants 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 and the coarse-grained call graph building algorithm (identical to CHA, see Tip et al. [66]). Dynamic control and data-flow analyses are supported to a certain extent.
- Fujaba [52] decomposes patterns into elemental design patterns (see [61, 36, 60]), which enables recognition of rather non-standard pattern occurrences.

3.3. Evaluation Methodology

Selected Patterns. In this evaluation precision, recall and speed were measured manually for the tools detecting the implementations of the following patterns:

- Collaborational: Decorator, Chain of Responsibility, Proxy, Observer, Composite, Visitor, State, Strategy, Bridge.
- Creational: Abstract Factory, Factory Method, Builder, Singleton, Prototype.

The reasons behind the selection of these patterns are:

- Covering a broad set of patterns presented in [27].

- Patterns are sufficiently complex for demonstrating the value of behavioural techniques.
- Many interesting implementation variants - for example, in the Observer pattern a subject might store observers in a collection field or in a finite number of scalar-typed fields.
- Challenge of discriminating the patterns that have identical structure but differ in behavior (e.g. Decorator vs. Chain of Responsibility).
- Challenge of using of additional information (e.g. program element names) to discriminate the patterns that have identical structure and behavior (e.g. Bridge vs. Strategy).

Several tens of design pattern occurrences were taken from the code repositories listed in Section 3.1 and from the repository of occurrences P-Mart `http://www.ptidej.net/download/pmart/`. All occurrences were thoroughly discussed with the authors of the reviewed tools.

Measuring of arruracy and speed. Speed measurements were carried out manually on a Dell Precision 370 desktop computer equipped with a 3,6 GHz quadcore CPU and 2 GB of RAM running Windows XP SP 3 and Eclipse Classic 3.5.2.

For each reviewed tool T, precision and recall were computed by comparing the mandatory role players in each candidate against the mandatory role players in the corresponding occurrence(s). Only roles *reported* by T are taken into account. For example, Ptidej [29] reports only class role players; therefore only class names were compared.

In order to compute the accuracy for each individual tool, candidates and occurrences were computed in the same way a given tool does. Take, for example, a Decorator candidate returned by SSA [68] that consists of a decorator class D, a component class C and two forwarding methods M_1 and M_2. Pinot [58] reports two candidates: one for each forwarding method. Accuracy is computed by searching for one (for SSA) and respectively, for two (for Pinot) identical occurrence(s) in the test set.

The results for each tool were compared and discussed with the authors of the corresponding tools.

3.4. Main Problems

This section discusses several major problems related to pattern detection that are tackled in this thesis. These problems were mentioned in the theoretical review of Dong et.al. [17] and in the evaluation done by Fülöp et al. [26] . The problems are confirmed by the practical evaluation of the pattern detection tools mentioned in Section 3.2 and are provided in the subsections of this section:

(1) Diversity of pattern detection results.
(2) Reasons behind false positives - insufficiently strong behavioral analyses.

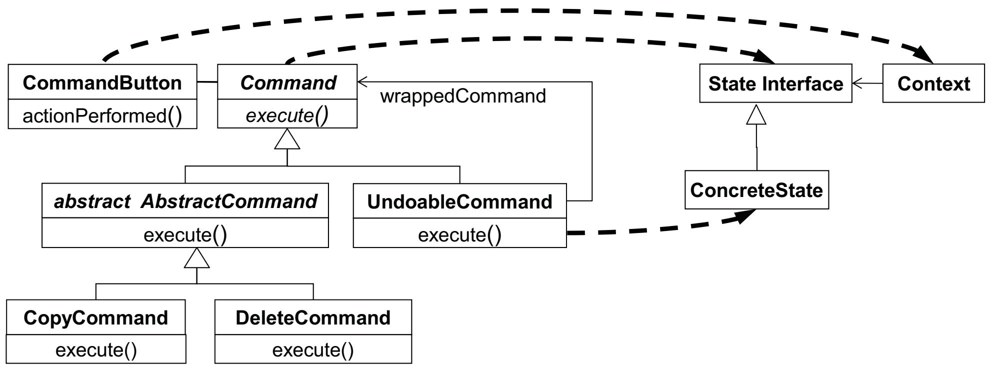


FIGURE 3.4.1. A Decorator candidate classified as State by PTIDEJ and FUJABA. Arrows illustrate the mapping of roles in a Decorator candidate into the roles of a State pattern.

(3) Reasons behind false negatives - too restrictive structural analyses.

Throughout this chapter, the abbreviation "CoR" is used when the Chain of Responsibility pattern is addressed.

Diversity of pattern detection results . It was observed that different pattern detection tools render different results on the same code by classifying pattern candidates as candidates of more general patterns. For example, Ptidej and Fujaba often classify Decorator candidates as State candidates. Figure 3.4.1 illustrates such a case. 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 (at some points) behaviorally: Decorator mimicks State behavior since control traverses several "concrete decorator" objects and finally reaches a "concrete component". This resembles the control flow in the State pattern when different states are switched. The two tools apparently 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).

In the Decorator candidate presented in Figure 3.4.1 the LEAF class is abstract. Therefore PINOT (which requires LEAF classes to be concrete) does not report Decorator. Additional experiments yielded more misclassifications of the same type that occur for certain implementation variants. During the analysis of the algorithms implemented in the reviewed pattern detection tools, it was observed that such misclassifications are caused by the fact that only a subset of the constraints defining a pattern holds for these variants.

It was observed that

- the number of true positives among simpler patterns (Adapter, Command, State, Strategy, Iterator, etc.) is much higher than

among more complex ones (Decorator, Chain of Responsibility, Observer etc.).
- the analysed tools give simpler patterns higher scores than they assign to more complex ones, usually 70-75% or higher;
- occurrences of simpler patterns are often classified identically by more than one tool (in around 80% cases).

Obviously, detection of simpler patterns is more reliable and the tools express their higher confidence in simpler patterns by the higher score. The main reason is that a smaller number of constraints must be satisfied. Furthermore, simpler design patterns give rise to less implementation variants. Elemental design patterns (EDPs, [35]) usually have only one variant since they are just *very* small patterns that contain less behavioral aspects so they are recognized with very high reliability and very high scores. In contrast, occurrences of big behavioral patterns are rarely classified identically by several pattern detection tools.[2] This happens mainly in the code that follows pedantically the implementation rules of the pattern descriptions of Gamma et al. [27].

Chapter 4 explains how to combine judgements of different tools in order to deduce correct results.

Reasons behind false positives. False positives occur in the pattern detection tools that do not support sufficiently strong behavioral analyses. Except for rare exotic implementations, most practical scenarios can be divided into several categories that are discussed in the following subsections. Each scenario is illustrated by example(s).

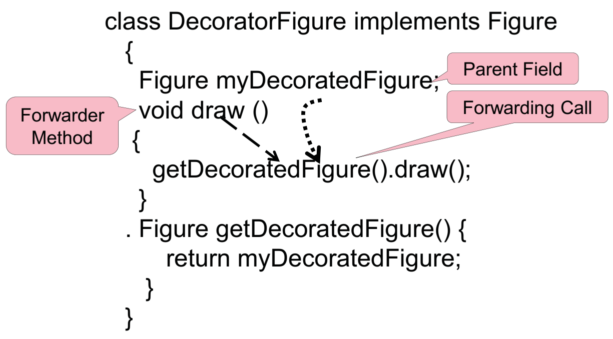
Forward to a single object. In the Decorator (respectively, the Proxy and the CoR) patterns the forwarding method of a decorator (respectively, proxy or handler) object invokes the forwarding method only on the *single* "parent" object (this invocation can be repeated more than once).

Figure 3.4.2a presents a Decorator occurrence from JHotDraw 6.0. The DECORATOR class is *DecoratorFigure*. The field $myDecoratedFigure$ is the PARENT FIELD and the RECEIVER VARIABLE. It points to the object that serves as the receiver of a forwarding call.

Figure 3.4.2b provides a false Decorator candidate taken from JHotDraw 6.0. The DECORATOR class is *FigureAndEnumerator*. Its fields $myFE1$ and $myFE2$ are PARENT FIELDS and RECEIVER VARS. Forwarding might be addressed at two receiver objects (stored in $myFE1$ and $myFE2$), i.e. in a "tree-like" way. Therefore such an implementation variant is referred to in the thesis as the *Tree Forwarder*.

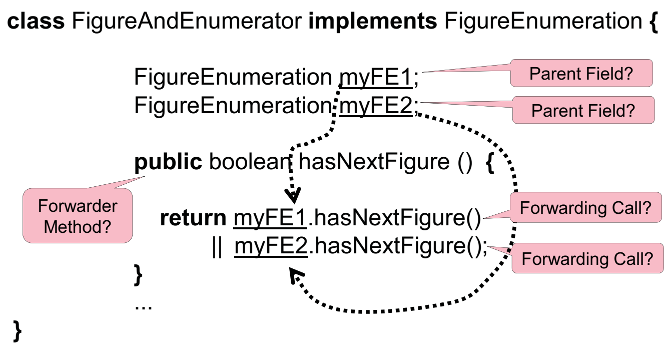
All tools that report method role players and recognize the Decorator pattern (PINOT and SSA) deliver two (false) Decorator candidates where $myFE1$ is at the same time the PARENT FIELD and the RECEIVER VAR and $myFE1.nextFigure()$ is the FORWARDING CALL.

[2]For instance, only 15% of the detected Visitor and 10% of the detected Bridge are classified identically.



```
class DecoratorFigure implements Figure
 {
  Figure myDecoratedFigure;
  void draw ()
  {
   getDecoratedFigure().draw();
  }
 . Figure getDecoratedFigure() {
     return myDecoratedFigure;
   }
 }
```

(A) Single object is addressed (JHotDRaw 6.0)



```
class FigureAndEnumerator implements FigureEnumeration {

          FigureEnumeration myFE1;
          FigureEnumeration myFE2;

          public boolean hasNextFigure () {

            return myFE1.hasNextFigure()
                || myFE2.hasNextFigure();
          }
          ...
}
```

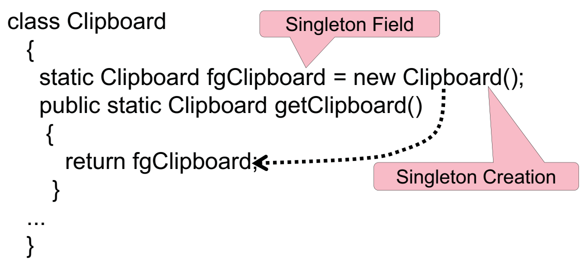
(B) Multiple Objects Addressed (JHotDraw 6.0)

Figure 3.4.2. Examples for forwarding to a single object

Keep all Singleton objects in shared variables. According to Stencel et al. [64], all newly created Singleton objects must eventually get into a finite number of shared variables. Figure 3.4.3a provides a Singleton pattern occurrence from JHotDraw 5.1. The class $Clipboard$ is the Singleton Type and the shared field $fgClipboard$ into which the singleton objects flow is the Singleton Variable.

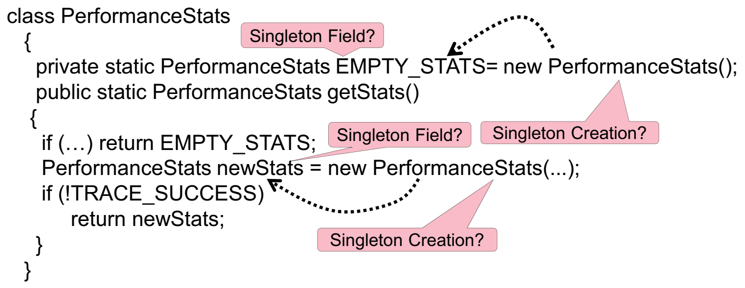
Figure 3.4.3b describes a false Singleton candidate from the Eclipse source code (the Runtime Core). The class $PerformanceStats$ is the Singleton Type, the variables $EMPTY\_STATS$ and $newStats$ are Singleton Variables and the objects created by $newPerformanceStats(" ", " ")$ and $new\ PerformanceStats(event, source)$ are the Singleton Objects. This Singleton candidate is false since the variable $newStats$ is not a shared variable and therefore any number of Singleton objects might be created.

Note that among the tools analyzed in this thesis, only D-Cubed (Stencel et al. [64]) checks whether all Singleton objects are kept in shared variables and thus correctly identifies the scenario in Figure 3.4.3b as a false Singleton candidate.



```
class Clipboard
 {
   static Clipboard fgClipboard = new Clipboard();
   public static Clipboard getClipboard()
    {
      return fgClipboard;
    }
  ...
 }
```

(A) All Singleton Objects in one shared variable (Lexi 0.1.1.alpha)



```
class PerformanceStats
 {
   private static PerformanceStats EMPTY_STATS= new PerformanceStats();
   public static PerformanceStats getStats()
   {
    if (...) return EMPTY_STATS;
    PerformanceStats newStats = new PerformanceStats(...);
    if (!TRACE_SUCCESS)
        return newStats;
   }
 }
```

(B) Singleton candidates are not in shared variables (Eclipse Runtime)

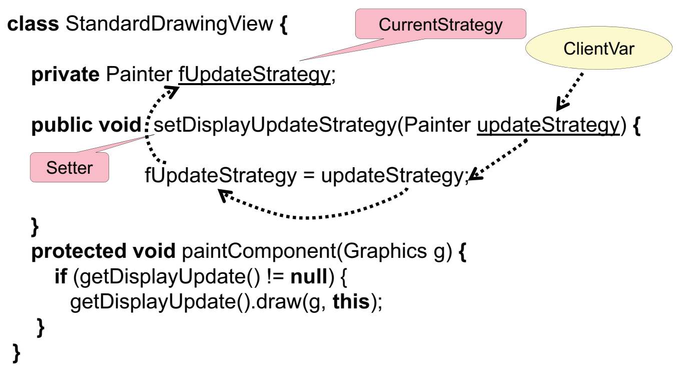
FIGURE 3.4.3. Examples demonstrating Singleton objects in shared variables

Updating relevant fields. Figure 3.4.4a provides a Strategy pattern occurrence from JHotDraw 6.0. The class *StandardDrawingView* is the strategy context. Its field *fUpdateStrategy* is the CURRENT STRATEGY and points to an object of the type *Painter*. *Painter* is the STRATEGY. The method *StandardDrawingView.setDisplayUpdateStrategy* causes updating of the field *fUpdateStrategy* after its owner is created.

In contrast, Figure 3.4.4b provides a false Strategy candidate from AWT 1.14 accepted by SSA [68], Fujaba [71] and Ptidej [50]. The class *RenderableImageOp* is the context. The field *myCRIF* is the CURRENT STRATEGY. It points to an object of the type *ContextualRenderedImageFactory*. This is the STRATEGY. The field *myCRIF* is updated only when its owner is created.

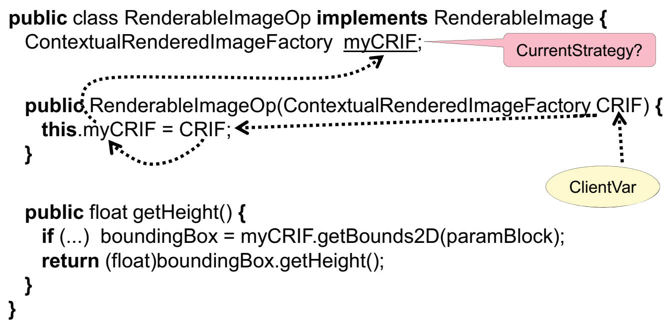
Propagating state changes. In the Observer pattern, subject state changes are propagated when a notification method causes the dataflow from the field(s) $sstate_1 ... sstate_s$ containing the subject state to the field(s) $ostate_1 ... ostate_o$ containing the observer state.

All practical scenarios where state change information is relevant are illustrated in the subsequent paragraphs:



```
class StandardDrawingView {

  private Painter fUpdateStrategy;

  public void setDisplayUpdateStrategy(Painter updateStrategy) {

         fUpdateStrategy = updateStrategy;

  }
  protected void paintComponent(Graphics g) {
     if (getDisplayUpdate() != null) {
       getDisplayUpdate().draw(g, this);
    }
}
```

(A) Relevant field is updated after construction of its owner(JHotDRaw 6.0)



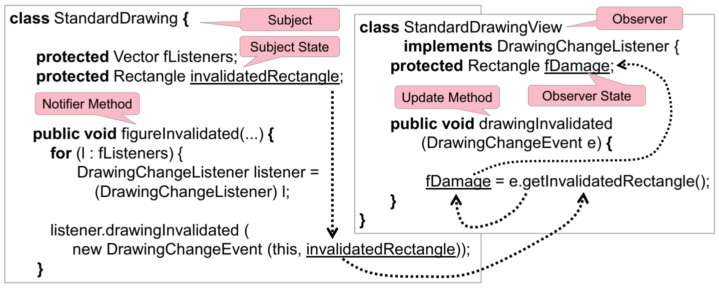
```
public class RenderableImageOp implements RenderableImage {
  ContextualRenderedImageFactory  myCRIF;

  public RenderableImageOp(ContextualRenderedImageFactory CRIF) {
    this.myCRIF = CRIF;
  }

  public float getHeight() {
    if (...)  boundingBox = myCRIF.getBounds2D(paramBlock);
    return (float)boundingBox.getHeight();
  }
}
```

(B) Relevant field is updated only during construction of its owner(AWT 1.14)

FIGURE 3.4.4. Updates of relevant fields

Scenario 1 - Propagating state changes in an Observer occurrence. Figure 3.4.5a describes an Observer occurrence from JHotDraw 5.1. The SUBJECT is the class *StandardDrawing*. Its field *fListeners* is the OBSERVERS VARIABLE. It points to the *drawing change listeners*, i.e. objects of the type *StandardDrawingView*. This interface is the OBSERVER and represents *drawing views* that depict some parts of drawing containers and must be notified if a drawing container is invalidated. The method *StandardDrawing.figureInvalidated* is the NOTIFIER. The field(s) *invalidatedRectangle* is the SUBJECT STATE and represent rectangular areas wrapping changed areas ("invalidated areas").

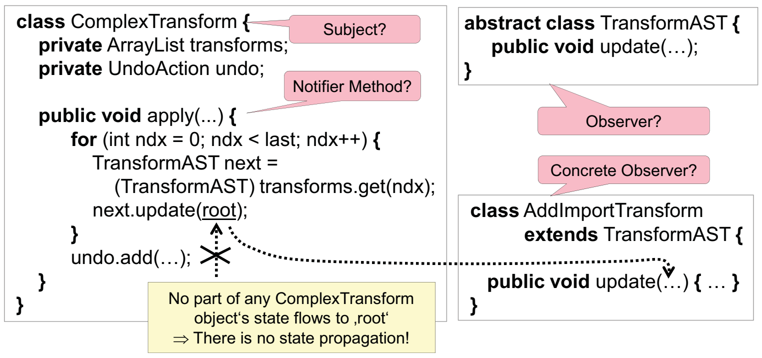
When a drawing container is changed, its "invalidated area" is updated; the updates must be propagated to each field(s) *o.fDamage* where *o* is the related drawing view (i.e. an object pointed to by*fListeners*). The field *fDamage* is the OBSERVER STATE and represents the rectangular area(s) in drawing views that were affected by changes in drawings.



```
class StandardDrawing {

    protected Vector fListeners;
    protected Rectangle invalidatedRectangle;

    public void figureInvalidated(...) {
      for (l : fListeners) {
          DrawingChangeListener listener =
            (DrawingChangeListener) l;

      listener.drawingInvalidated (
          new DrawingChangeEvent (this, invalidatedRectangle));
    }
```

```
class StandardDrawingView
      implements DrawingChangeListener {
    protected Rectangle fDamage;

    public void drawingInvalidated
          (DrawingChangeEvent e) {

          fDamage = e.getInvalidatedRectangle();
      }
}
```

(A) Data Flow in the Observer pattern (JHotDRaw 5.1)



```
class ComplexTransform {
    private ArrayList transforms;
    private UndoAction undo;

    public void apply(...) {
        for (int ndx = 0; ndx < last; ndx++) {
            TransformAST next =
                (TransformAST) transforms.get(ndx);
            next.update(root);
        }
        undo.add(...);
    }
}
```

```
abstract class TransformAST {
    public void update(...);
}
```

```
class AddImportTransform
            extends TransformAST {

    public void update(...) { ... }
}
```

Subject?
Notifier Method?
Observer?
Concrete Observer?
No part of any ComplexTransform object's state flows to ‚root'
⇒ There is no state propagation!

(B) A service loop (JRefactory 2.6)

FIGURE 3.4.5. Propagating state changes

The changes in the state of the subject are propagated to each of its observer via the dataflow from $StandardDrawingView.invalidatedRectangle$ into $o.fDamage$. This dataflow is caused by the actions of N .

Scenario 2 - No data flow in a false candidate. The second scenario, provided in Figure 3.4.5b describes a false Observer candidate from JRefactory 2.6 accepted by SSA [68], Fujaba [71] and Ptidej [50]. The Subject is $ComplexTransform$ that represents *complex transformations*. The field $transforms$ is the OBSERVERS VARIABLE. It points to *basic* transformations (i.e. instances of $TransformAST$) that comprise a complex transformation. $TransformAST$ is the OBSERVER. Basic transformations are executed sequentially via the NOTIFICATION METHOD *apply*. This, in turn, calls the method *b.update* of each basic transformation *b*. There is no concept of "subject state" or " observer state" and no relevant dataflow.

In general, in a false Observer candidate a master object can own some method that iterates over a collection of dependent objects in order to perform some job. Such candidates (like the one provided in Figure 3.4.5b) are referred to as *service loops*.

Control and Data Flow Order. Assume that an Observer occurrence includes a subject S and a notifier method $S.N$. Denote the SUBJECT STATE fields by $S.f_1...S.f_k$. According to [27] the dataflow into $S.f_1...S.f_k$ caused by the actions of $S.N$ must not occur after at least one update call is issued by $S.N$, i.e. occurs in *the sequential order*.

The order of control and data flow is restricted also in other design patterns. Take, for example, the State pattern whose behavior is investigated by Wendehals [75]. Denote the "state machine" method by M_S, the related state action method by A and the method(s) causing the update of the current state variable by M_U. Within the control flow of M_S, A and M_U must be invoked in turn, and in addition some M_U must precede A in order to show that the current state must be initialized, then some actions should be executed (i.e. *the interleaving order*).

It should be noted that expressing control / data flow order by means of regular expressions is not a mere occassion. Poch et al. [55] argue that sequences of relevant actions in legacy applications can be specified by finite automata.

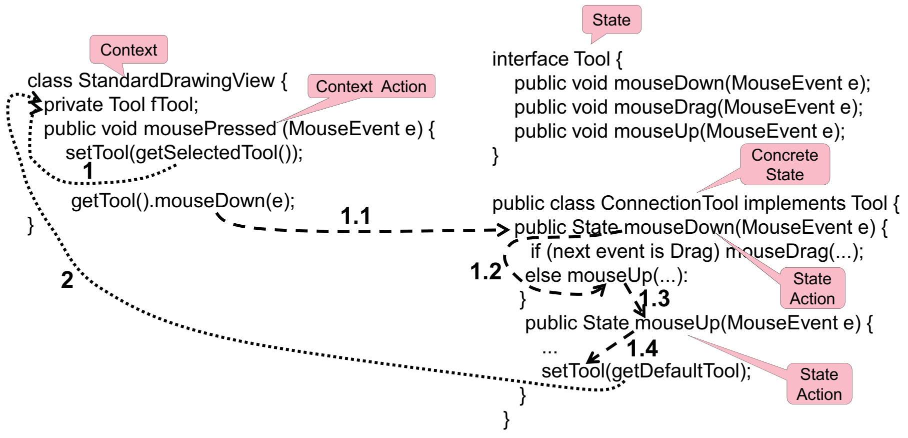
The interleaving order is demonstrated on a State pattern candidate from JHotDraw 6.0 that is provided in Figure 3.4.6a. The CONTEXT is the class $StandardDrawingView$. The CURRENT STATE field is $fTool$ that points to the objects (concrete states) belonging to the type $Tool$.

The LAUNCHER method is $StandardDrawingView.mousePressed$ that is invoked when a user selects a tool (connection, fill color etc). The initial state is assigned to the selected tool and its ACTION method $mouseDown(...)$ is called. Then several "mouse drag" events (corresponding to the action methods $mouseDrag$) are handled. Finally, when the mouse is released, the action method $mouseUp$ is called, followed by the resetting the current state (i.e. the current tool) to the default one. Thus, several action methods reside between two updates of the current state.

Figure 3.4.6b provides a Strategy occurrence from AWT 1.14 where the interleaving order does not occur. The class $CheckBox$ is the CONTEXT. The field $listener$ points to instances of the class $ItemListener$ that is STATE. The STATE ACTION method $ItemListener.itemChanged(Event\, e)$ accesses the field $listener$. The method $CheckBox.addListener()$ that modifies the field $listener$ after an instance of $CheckBox$ is constructed (since State and Strategy are decoupler patterns, see Par. 2.4) might be called at any time (independently of any call to $itemChanged(Event\, e)$).

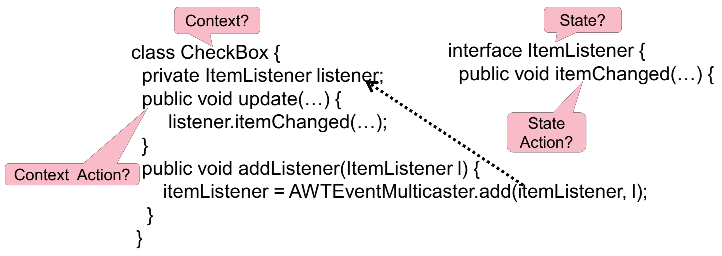
This candidate was erroneously marked as State by PINOT [58] and Fujaba [71] since these tools do not take the order between state actions and state variable updates into account. Ptidej [50] supports the interleaving order by examining execution traces and recognizes this candidate properly, as a Strategy candidate.

Detecting various Proxy implementations. In a Proxy occurrence, a real subject must have the type that is a sibling of the proxy type. In the "classical" implementation variant (*Statically Typed Proxy*) this requirement is enforced by defining the static type of the Subject field as a sibling of the proxy. In the *Dynamically Typed Proxy* the static type of the Subject field



```
class StandardDrawingView {
  private Tool fTool;
  public void mousePressed (MouseEvent e) {
    setTool(getSelectedTool());

    getTool().mouseDown(e);
}

interface Tool {
   public void mouseDown(MouseEvent e);
   public void mouseDrag(MouseEvent e);
   public void mouseUp(MouseEvent e);
}

public class ConnectionTool implements Tool {
   public State mouseDown(MouseEvent e) {
    if (next event is Drag) mouseDrag(...);
    else mouseUp(...):
   }
   public State mouseUp(MouseEvent e) {
     ...
     setTool(getDefaultTool);
   }
}
```

(A) The interleaving order (JHotDRaw 6.0)



```
class CheckBox {
  private ItemListener listener;
  public void update(...) {
     listener.itemChanged(...);
  }
  public void addListener(ItemListener l) {
     itemListener = AWTEventMulticaster.add(itemListener, l);
  }
}

interface ItemListener {
  public void itemChanged(...) {
```

(B) No interleaving order in Strategy (AWT 1.14)

FIGURE 3.4.6. Ordered Control and Data flow

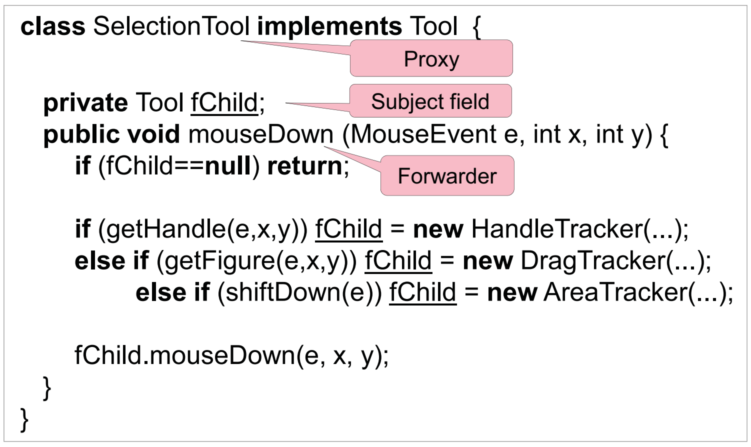
is a supertype of the proxy type. In this case, the subject field is assigned at runtime to a real subject object(s) whose type is a sibling of the Proxy object type.

Figure 3.4.7a provides a Dynamically Typed Proxy occurrence from JHotDraw 5.1. The PROXY is $SelectionTool$. The field $fChild$ is the SUBJECT. A forwarding method $SelectionTool.mouseDown(...)$ initializes the points-to set of $fChild$ to the set $\{newHandleTracker(...), newAreaTracker(...), newDragTracker(...)\}$. These objects are Real Subjects and the type of each one is a sibling of $SelectionTool$. Other forwarding methods ($mouseDrag(...)$ and $mouseUp(...)$) invoke the corresponding methods of real subjects.

Figure 3.4.7b provides a Decorator occurrence from JHotDraw 5.1. Here forwarding can be done to any object belonging to a subclass of Decorator.

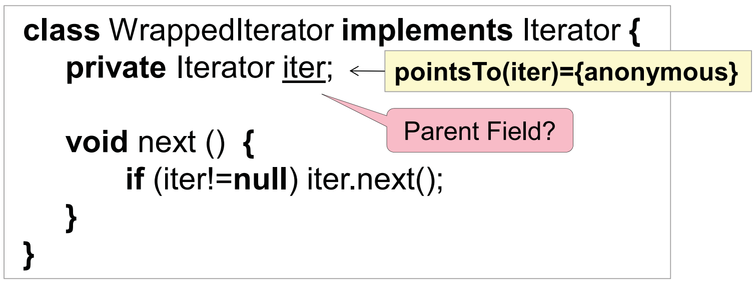
No pattern detection tool mentioned in Section 3.2 detects Dynamically Typed Proxy candidates.

Object Coverage by Calls. Certain collaboration patterns (Observer, Composite, CoR, Decorator) restrict the set of objects pointed to by a set of relevant fields $f_1..f_n$ on which some callee method m is invoked.



```
class SelectionTool implements Tool {

  private Tool fChild;
  public void mouseDown (MouseEvent e, int x, int y) {
    if (fChild==null) return;

    if (getHandle(e,x,y)) fChild = new HandleTracker(...);
    else if (getFigure(e,x,y)) fChild = new DragTracker(...);
         else if (shiftDown(e)) fChild = new AreaTracker(...);

    fChild.mouseDown(e, x, y);
  }
}
```

(A) Dynamically Typed Proxy



```
class WrappedIterator implements Iterator {
   private Iterator iter;

   void next () {
       if (iter!=null) iter.next();
   }
}
```

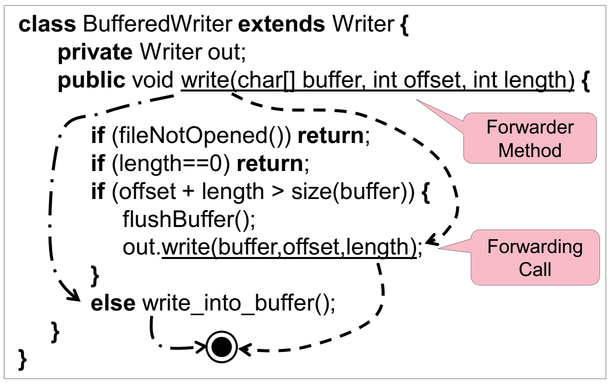
(B) Decorator

FIGURE 3.4.7. Various ways of forwarding

All practical scenarios where object coverage is relevant can be divided into several categories provided in the following paragraphs. In the figures provided in these paragraphs, the relevant method calls are underlined and the end point of a method is depicted by the black colored ball. Note that only PINOT restricts (in a limited way and only in certain patterns) the set of objects on which a callee method(s) are invoked.

Scenario 1 - PINOT reports a correct diagnosis. Figure 3.4.8 illustrates a CoR occurrence from Java IO accepted by PINOT [58]. The class *BufferedWriter* is the CONCRETE HANDLER. The field *out* is the NEXT FIELD and the method *write* is the FORWARDER. This is a CoR occurrence since *write* forwards *conditionally*, i.e. only when the write request causes moving beyond the buffer boundaries.[3] The corresponding execution is shown by the thick dotted-dashed line.

[3]In contrast, in a Decorator occurrence forwarding is done *unconditionally*, i.e. in each execution



```
class BufferedWriter extends Writer {
    private Writer out;
    public void write(char[] buffer, int offset, int length) {

        if (fileNotOpened()) return;
        if (length==0) return;
        if (offset + length > size(buffer)) {
            flushBuffer();
            out.write(buffer,offset,length);
        }
        else write_into_buffer();
    }
}
```

FIGURE 3.4.8. Partial Coverage is satisfied in a CoR occurrence(Java IO 1.4)

It should be noted that PINOT uses only a limited information about the control flow and might deliver false positives when applying its simplified variant of the Object Coverage concept. The next scenario illustrates such a case.

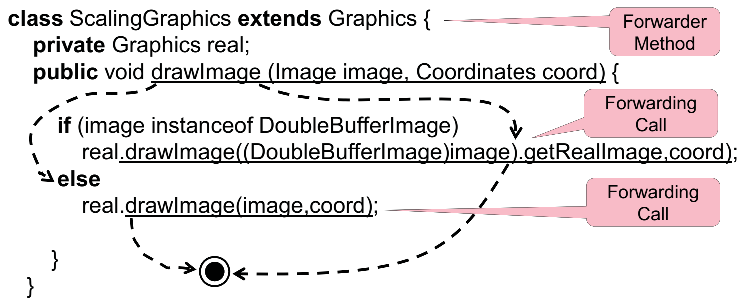
Scenario 2 - PINOT reports an incorrect diagnosis because of the limitation of its "conditional forwarding" concept. Figure 3.4.9 provides a Decorator occurrence from JHotDRaw 6.0. The class *ScalingGraphics* is the DECORATOR. The field *real* is the NEXT FIELD and the method *drawImage*(...) is the FORWARDER. The forwarding is done in both branches of the corresponding if-statement, and, consequently, in *each execution* of *drawImage*(...).

PINOT [58] erroneously reports a CoR candidate where the field *real* is the NEXT FIELD and the method *drawImage* is the FORWARDER. This is because PINOT finds the forwarding call in *some* branch of the if-statement and therefore (erroneously, *without examining another branch*) decides that forwarding is done conditionally.

Note that PINOT uses the conditional invocation not in all relevant patterns. The next paragraph illustrates a false Observer candidate accepted by PINOT because the conditional invocation is not checked at all.

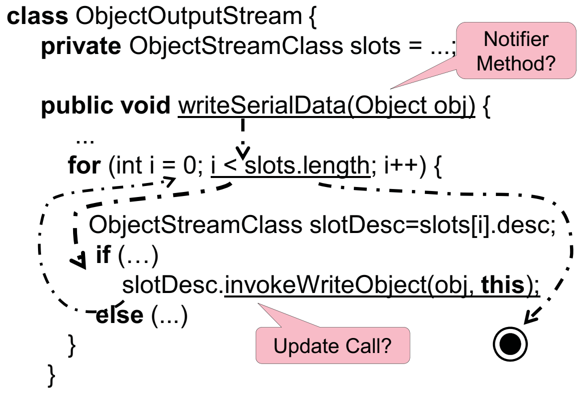
Scenario 3 - PINOT reports an incorrect diagnosis because the set of objects on which a callee method(s) are invoked is not restricted in a given patterns. Figure 3.4.10 illustrates a service loop from Java IO. A master *ObjectOutputStream* iterates over only those dependent objects (stored in its field *O.slots*) that belong to the type *ObjectStreamClass*. That is, the data is actually written only into the memory slots which support the writing action. The method *writeSerialData*(...) performs this iteration, invoking the method *invokeWriteObject* on each appropriate object.

PINOT erroneously reports an Observer candidate where the method *writeSerialData* is the NOTIFIER METHOD and the method *invokeWriteObject*



```
class ScalingGraphics extends Graphics {
  private Graphics real;
  public void drawImage (Image image, Coordinates coord) {

    if (image instanceof DoubleBufferImage)
      real.drawImage((DoubleBufferImage)image).getRealImage,coord);
    else
      real.drawImage(image,coord);

  }
}
```

FIGURE 3.4.9. Total Coverage is satisfied in Decorator (JHotDRaw 6.0)



```
class ObjectOutputStream {
  private ObjectStreamClass slots = ...;

  public void writeSerialData(Object obj) {
    ...
    for (int i = 0; i < slots.length; i++) {
      ObjectStreamClass slotDesc=slots[i].desc;
      if (...)
        slotDesc.invokeWriteObject(obj, this);
      else (...)
    }
  }
```

FIGURE 3.4.10. Partial Coverage is violated in a false observer candidate (Java IO 1.4)

is the UPDATE METHOD. This is a false candidate since it might happen that the method *invokeWriteObject* is not invoked on each dependent object within a loop.

Types of Created Objects. Following Gamma et al. [27], the creation of a distinct "product family" can be implemented in the creational patterns in two following ways:

(1) Override a creator method. An "abstract creator" class is defined. "Creator method(s)" are overridden in its subclasses. Creator methods with the same signatures belonging to the different concrete factories create products of different types. This implementation variant was mentioned by Eden et al. [19].
(2) Parameterize a creator method. The values of parameters passed to creator method calls determine the type of a created product. This implementation variant was mentioned by Gamma et al. [27, page 126].

An Abstract Factory occurrence from JRefactory 2.6 (accepted by PINOT and Ptidej) is provided in Figure 3.4.11a. The ABSTRACT FACTORIES are $BufferParserFactory$ and $FileParserFactory$. Their methods $getInputStream()$ and $getInputStream()$ are CREATOR METHODS. These methods create products of the (different) types $FileInputStream$ and $BufferInputStream$ respectively. These are the CONCRETE PRODUCTS. The creator methods with the same signature that belong to different concrete factories create products of different types.

A false Factory Method candidate from JHotDRaw 6.0 (erroneously accepted by PINOT and Ptidej) is provided in Figure 3.4.11b. The FACTORies are *AbstractFigure* (*ComponentFigure* and *RectangleFigure*). The methods *ComponentFigure.handles*() and *RectangleFigure.handles*() are the FACTORY METHODS, creating products of the same type *HandleEnumerator* which is the CONCRETE PRODUCT.

This is a false candidate since, according to the logic of "factory" patterns, two different creator methods with the same signature must be responsible for creating products of different types (which is not the case here).

No pattern detection tool mentioned in Chapter 3 supports concepts similar to the one presented by the uniquely created product type.

The reasons behind false negatives. The reasons behind false negatives observed in this thesis are provided in the subsections of this section:

(1) Insufficient treatment of the transitivity of certain relations
(2) Too strong restrictions on the access qualifiers of relevant role players.
(3) Some relevant role players that reside in other repositories.

Insufficient treatment of transitivity. These false negatives occur because pattern tools do not sufficiently handle the transitivity of the following relations:

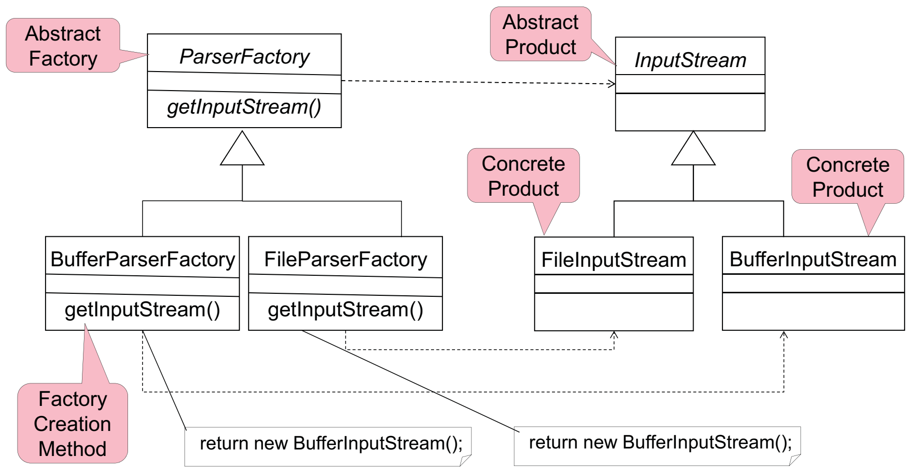
(1) Subtyping (intermediate classes in the subclassing relation)
(2) Method invocation (the presence of intermediate methods)

PINOT and DP-Miner do not support transitivity at all, thus supporting only small subsets of implementation variants provided in Gamma et al. [27]. SSA, Ptidej and Fujaba tolerate a limited case of the subtype transitivity (up to one intermediate class is supported). That is, scenarios like the one provided in Figure 3.4.14 are accepted since there is only one intermediate class (*AbstractFigure*). However, scenarios like the one provided in Figure 3.4.15 are already missed since more than one intermediate class exists.

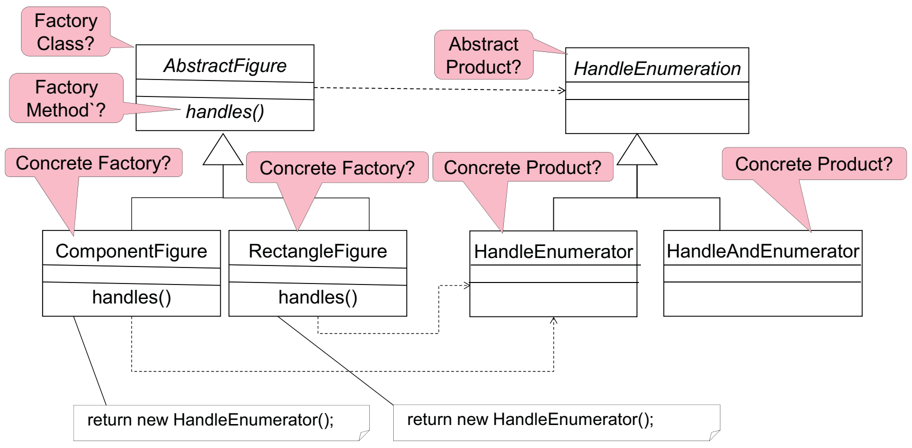
No tool reviewed in this thesis supports method invocation transitivity. That is, scenarios like the one provided in Figure 3.4.12a are not accepted.

The following paragraph describes the practical scenarios where the method invocation transitivity is involved.

Control Flow Scope. When a caller method M invokes other method(s), the corresponding control flow might go through some intermediate method(s)



(A) Product types are different (JRefactory 2.6)

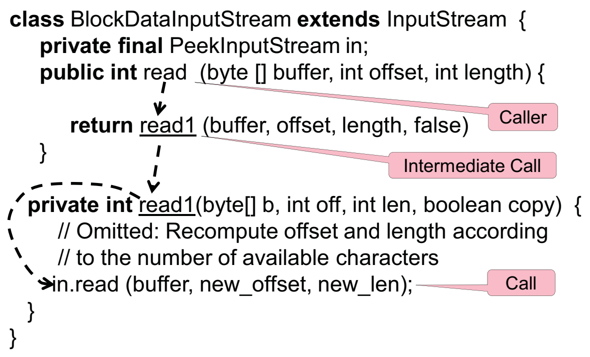


(B) Product types are the same (JHotDraw 6.0)

FIGURE 3.4.11. Creation of product objects

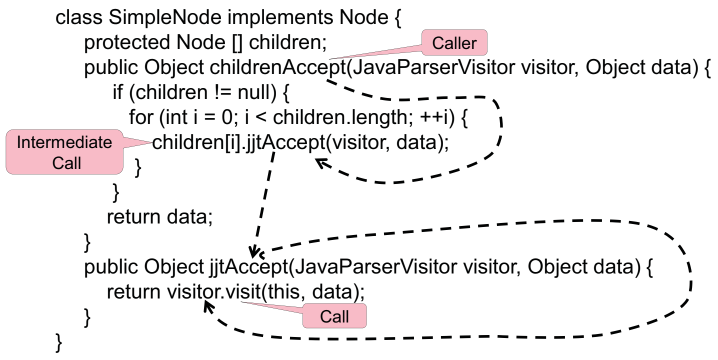
(Section 2.5). Following transitive control flow in an unbounded way can lead to large control flow relations, especially when analyzing real industrial code. Different patterns place certain restrictions on the set of objects owning intermediate methods.

Denote the object owning M by O. In all patterns (except Visitor) the receiver variables of intermediate method calls point to O; in this case the *intra-object control flow scope* takes place. In the Visitor pattern, a visit method call is control flow reachable from some accept method owned by an element object E. The receiver variables of intermediate method invocations might point not only to E but also to some other objects belonging to "element" type. In this case the *inter-object control flow scope* takes place.



```
class BlockDataInputStream extends InputStream {
    private final PeekInputStream in;
    public int read  (byte [] buffer, int offset, int length) {

        return read1 (buffer, offset, length, false)
    }
   private int read1(byte[] b, int off, int len, boolean copy)  {
       // Omitted: Recompute offset and length according
       // to the number of available characters
      in.read (buffer, new_offset, new_len);
   }
}
```

(A) Intra-Object Control Flow Scope (Java Io 1.4)



```
class SimpleNode implements Node {
    protected Node [] children;
    public Object childrenAccept(JavaParserVisitor visitor, Object data) {
        if (children != null) {
           for (int i = 0; i < children.length; ++i) {
              children[i].jjtAccept(visitor, data);
           }
        }
       return data;
    }
    public Object jjtAccept(JavaParserVisitor visitor, Object data) {
       return visitor.visit(this, data);
    }
}
```

(B) Inter-Object Control Flow Scope (JRefactory 2.6)

FIGURE 3.4.12. Control Flow Scope

Figure 3.4.12a illustrates a Change Of Responsibility occurrence from Java IO with the intra-object control flow scope. The forwarder method $public\,int\,read(byte[], int, int)$ invokes the forwarding call $in.read(buffer, new\_offset, new\_len)$ through the private intermediate method
$read(byte[], int, int, boolean)$.

Figure 3.4.12b illustrates a Visitor occurrence from JRefactory 2.6 with the inter-object control flow scope. The objects of the type $Node$ are the ELEMENTS. The ACCEPT METHODS are $childrenAccept(...)$ which invoke visit calls $v.visit(this, data)$ indirectly. The intermediate method is $jjtAccept(...)$. The receiver variables of the intermediate method invocations $children[i].accept(visit, param)$ point to the ELEMENT objects.

Too strong restrictions on access qualifiers. Access qualifier relaxation is supported by SSA, Ptidej and Fujaba. The roles that are supposed to be

```
public class Iconkit {
  private static Iconkit fgIconkit = null;
  public Iconkit(Component component) { ... }
   /** Gets the single instance */
  public static Iconkit instance() {return fgIconkit; }
...
}
```

FIGURE 3.4.13. Need to relax static constraints: A Singleton with a public constructor (JHotDraw 6.0)

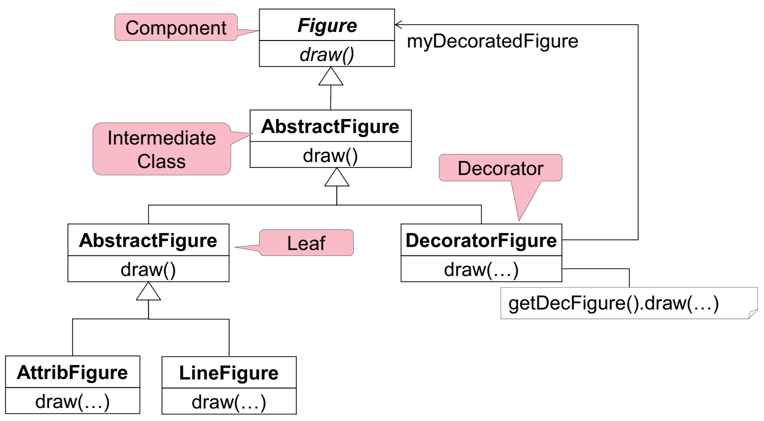


FIGURE 3.4.14. One intermediate class in subtyping (JHot-DRaw 6.0)

played by abstract classes / methods can be played by concrete classes or methods. For example, the Singleton occurrence scenario provided in Figure 3.4.13 is not accepted by PINOT because the constructor is public (PINOT requires a Singleton constructor to be private or protected) but is accepted by SSA, Ptidej and Fujaba.

Relevant role players belong to other repositories. In some pattern occurrences the relevant role players belong to the external repositories (usually to the external libraries like Java core). For example, in the Proxy occurrence from JHotDraw 6.0 illustrated in Figure 3.4.16 the the REAL SUBJECT and the SUBJECT (*List* and *Vector* respectively) belong to the Java core library. SSA, DP-Miner do not accept candidates whose role players belong to external repositories unless these are compiled and their byte-code is loaded into the tool. PINOT and Ptidej, in contrast, accept such a candidate by examining the references to the *List* and *Vector* classes and assuming that the missing parts "exist somewhere".[4]

[4]These are so-called "ghost elements" in Ptidej

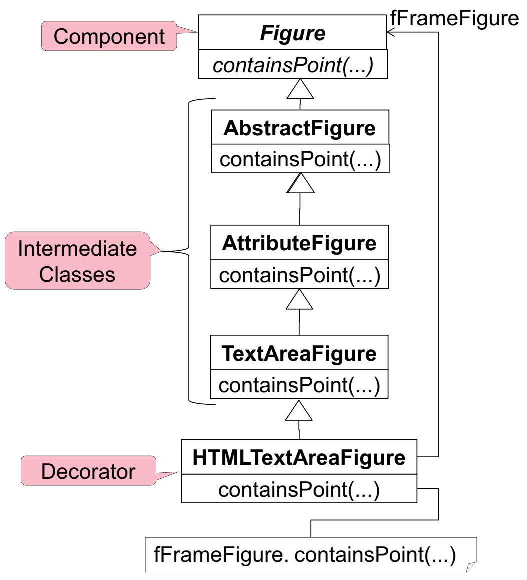


FIGURE 3.4.15. More than one intermediate class (JHotDraw 6.0)

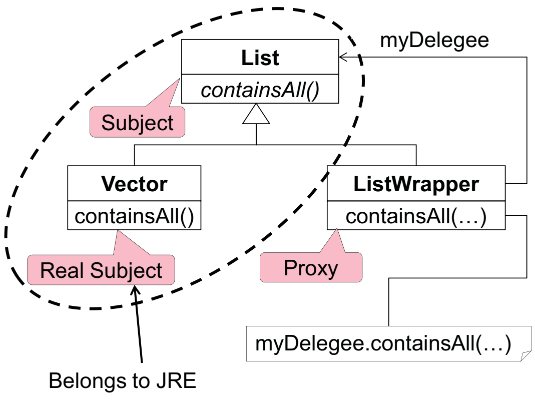


FIGURE 3.4.16. Some role players reside in external libraries

3.5. Conclusions

Pattern detection tools use various formalisms and concepts to describe their approaches and use different repositories to validate their results. Therefore different, contradictory results on the same code are returned.

Most differences occur because many behavioural aspects of design patterns that are originally expressed by Gamma et al. [27] *in the terms of interactions between objects*, are approximated by pattern tools differently at the program element level.

The research community needs a commonly agreed, domain-independent platform to reason about design patterns. This platform must provide a common basis so that researchers can express formally why a particular candidate is indeed an occurrence of a certain pattern. As a first step in this direction, it is decided to reason about the interactions within pattern occurrences in the terms of interactions between objects. The next chapter advocates this approach.

Having several pattern detection tools that report diverse results, another way to improve the overall accuracy is combining the outputs of these tools. Each tool, analyzing certain aspects of a pattern, contributes its estimates of which program elements are likely to form a particular pattern occurrence. This is the *data fusion approach to pattern detection* which is presented in Chapter 4.

Part 2

Approach 1: Data Fusion

CHAPTER 4

Data Fusion Approach

Diversity of results obtained from pattern detection tools (Section 3.4) inspired the following suggestion: Instead of striving for the ideal pattern detection tool by adding new techniques and–or upgrading incomplete ones, it is more promising *to fuse the outputs of existing tools*. Each tool, analyzing certain aspects of a pattern, contributes its estimates about which patterns are likely to be relevant. The *data fusion approach* builds thus on the synergy of proven techniques without requiring any expensive reimplementation of what is already available. Poshyvanyk et. al. [56] observed the advantages of fusing of outputs of several program comprehension tools that employ static and dynamic program analyses. Conservative but potentially imprecise static analyses can be compensated by precise dynamic analyses that do not cover all program executions.

Section 4.1 provides a brief overview of the data fusion approach for design pattern detection. Section 4.2 explains how structurally simpler patterns indicate the presence of more complex ones. Section 4.3 and Section 4.4 present the steps of the data fusion approach. Section 4.5 discusses the case when even simple patterns are misrecognized. Section 4.6 suggests additional ways to improve the accuracy of the data fusion approach. Section 4.7 presents the evaluation results. Finally, Section 4.8 presents the conclusions of the chapter.

4.1. Brief Overview over the Approach

It was noted in Section 3.4 that candidates for complicated patterns (e.g. Observer, Visitor) are rarely classified identically by several tools; when this occurs, a candidate is very likely to be a true positive.

More often, different pattern detection tools report simpler patterns that resemble more complicated ones structurally and-or behaviorally (e.g. Command instead of Visitor) , “agreeing” on smaller patterns. Smaller patterns are classified with high scores, high precision and identical role assignments for mandatory roles. Put another way, occurrences of smaller patterns *witness* occurrences of bigger patterns by virtue of being their substructures (see Section 4.2 for more details).

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 candidate C identically, the next stage starts - checking whether different tools identically classify "witnesses" of C and combining the corresponding judgements, as it is explained in Section 4.4.

4.2. Smaller patterns witness bigger patterns

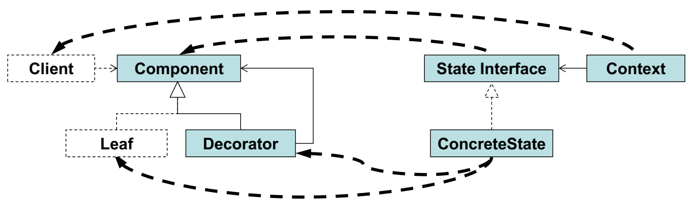
In general a pattern, B is *a subpattern* of another pattern A if all occurrences of A are occurrences of B - that is, all of the roles of B can be mapped to roles of A. In this case A is said to be a *superpattern* of B and B is a *subpattern* of A. Superpatterns are characterized by smaller number of constraints than their subpatterns and therefore are often detected instead of their subpatterns. Note that the subpattern relation is purely technical and **does not imply that the patterns have the same intent**.

A *witness* W of another pattern P is a pattern of which it is known that some or all of its roles also play some role in P. Thus an occurrence of a witness hints that there might exist an occurrence of the witnessed pattern. This is a rather general definition. In this thesis a specific form of witnesses is taken: superpatterns. They are taken by leveraging on the good recognition of simpler patterns.

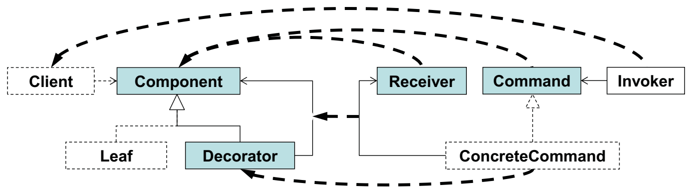
For example, Command and State are witnesses for Decorator because:

- **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 occurrence. Such behavior corresponds to the behavior of the Command pattern. A client of Decorator is the Command Invoker , the Component class corresponds to the Command Interface (and at the same time to the Receiver), the Decorator class corresponds to the Concrete Command role. See Figure 4.2.1b.
- **State**. When a Decorator occurrence runs, 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 class playing the Component role stands for the State Interface role, the invoker of Decorator plays the State Context role, Concrete Decorator and Leaf classes play the Concrete State role. See 4.2.1a.

Figures 4.2.1b and 4.2.1a illustrate that, because superpatterns are taken as witnesses for subpatterns, the witness role mappings result from reversing the edge directions in the corresponding subpattern role mappings. This has the implication that a particular design pattern recognition tool might report non-unique witness role mappings. For example, the Concrete State role is mapped to several Decorator roles.



(A) State is a witness of Decorator



(B) Command is a witness of Decorator. Role mappings are illustrated by arrows

FIGURE 4.2.1. Witnesses of Decorator

The problem of non-unique witness role mappings disappears only if several design pattern recognition tools correctly recognize a pattern occurrence, delivering identical role assignments (Section 4.3). Otherwise, in order to distinguish non-unique role mappings diagnostics of different witnesses are obtained from are combined (Section 4.4).

4.3. Agreement and Joint Recognition

If two design pattern recognition tools A and B believe that the same set of program elements is an occurrence C of the same pattern with sufficiently high scores (more than 50%) and their role assignments for the mandatory roles are the same, A and B are said to *agree* on C.

Figure 4.3.1 presents a Decorator occurrence from the *java.io* package on which PTIDEJ, SSA and PINOT agree with 50% score from Ptidej, 60% from SSA and 100% from PINOT[1]. 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 occurrence of a pattern P and other tools do not disagree on P's witnesses, it is said that this pattern occurrence is *jointly recognized*. The Decorator occurrence from *java.io* (Figure 4.3.1) was jointly recognized by SSA, PINOT and PTIDEJ.

If tools do not agree on a candidate of a complex pattern P, one searches for the agreement on the witnesses of P. Then it is possible to *reconstruct the pattern from witnesses*, as described in Section 4.4.

[1]FUJABA and DP-Miner do not recognize decorators at all.

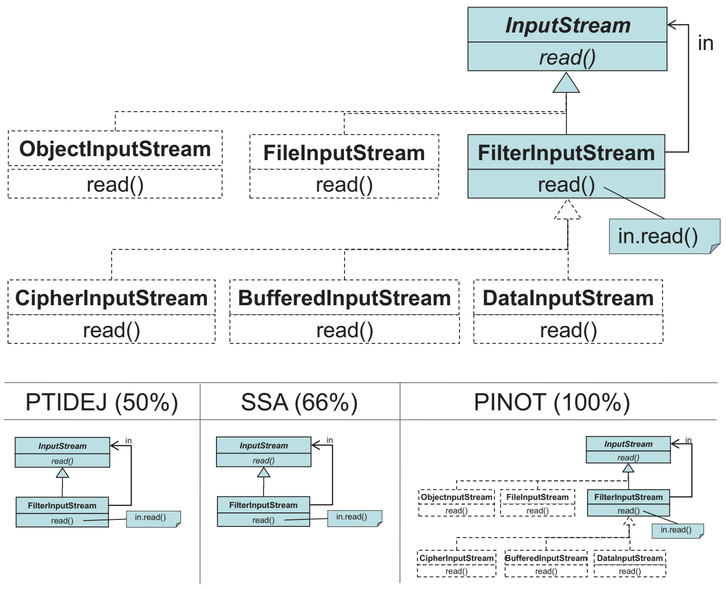


FIGURE 4.3.1. Three tools agree on this Decorator occurrence

4.4. Disagreement and Reconstruction

Unfortunately, different tools agree on occurrences of complex patterns like Decorator quite rarely - mostly this occurs in straightforwardly implemented occurrences from well-designed repositories like JUnit. However, it was observed that in most cases tools disagree on big pattern occurrences but agree on most of their witnesses. For example, in JHotDraw 6.0 the evaluated tools disagree on more than 70% of all Decorator occurrences but agree on around 85% of Decorator witnesses.

Let different tools agree with at least 66% score on occurrences of patterns that witness a pattern P. If the related role assignments and witness role mappings for all the witnessing occurrences are complete for P and consistent it is said 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. A role mapping is *complete* for P, if all the mandatory roles of P are assigned to some elements.

Figure 4.4.1 presents a Decorator occurrence from JHotDraw 6.0 on which the tools disagree. However, this Decorator occurrence can be reconstructed from witnesses because three tools agree on the existence of witnesses among the program elements in Figure 4.4.1.

Fujaba and DP-Miner do not support Decorator at all. PINOT rejects this occurrence since the implementation variant in which the LEAF role is played by an abstract class (*AbstractCommand*) instead of a concrete class is not accepted. Fujaba rejects this occurrence since the Decorator-Component

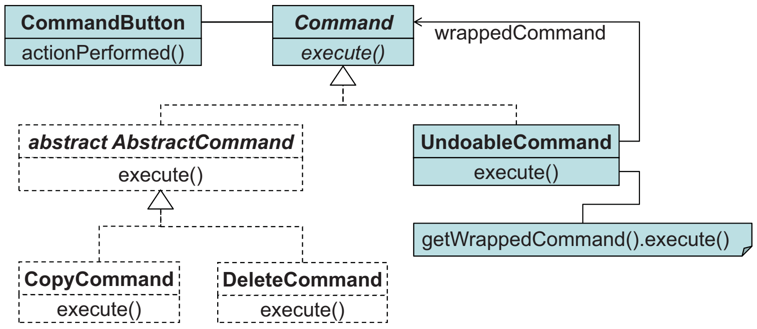


FIGURE 4.4.1. Tools disagree on this Decorator occurrence

delegation is required to be done directly through a field; Ptidej lacks behavioral analyses to recognize the getter method $getWrappedCommand()$. Only SSA identifies the pattern thanks to the appropriate liberal criteria and yields a consistent role mapping. So, each tool rejects this Decorator occurrence due to its individual reason - unwillingness to accept a pattern, certain implementation variants or lacking behavioral analyses.

Although the tools disagree on this Decorator occurrence, it can be reconstructed because three tools agree on the existence of the witnesses among the program elements in Figure 3.4.1:

- Fujaba and Ptidej agree on an occurrence of State with 80% score from FUJABA and 100% score from Ptidej. Both agree that the COMMAND role maps to the STATE INTERFACE role whereas $UndoableCommand$, $CopyCommand, DeleteCommand$ are the CONCRETE STATES role. *CommandButton* is the CONTEXT.
- SSA and Ptidej agree on an occurrence of the Command pattern with 25% score from Ptidej and 100% score from SSA. Both agree that *Command* is the COMMAND INTERFACE and the RECEIVER whereas *UndoableCommand*, *CopyCommand*, *DeleteCommand* are the CONCRETE COMMANDS and *CommandButton* is the INVOKER.
- PINOT and Fujaba agree on an occurrence 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 (see Figures 4.2.1a, 4.2.1b). The scores assigned to the recognized subpatterns are high.

It was observed that for correctly identified pattern occurrences Ptidej might return low scores (the Command witness got just 25%). Discussions with authors of the analyzed tools revealed that different tools have different ways to weigh relationships. So scores from different tools may not be comparable. That is, the scores of individual tools must be adjusted in order to compensate their different degree of optimism and pessimism. This issue is addressed by the common exchange format for pattern detection tools presented in Chapter 5.

4.5. Wrongly Identified Superpatterns

Sometimes pattern detection tools fail to correctly identify a superpattern and may even agree on wrong diagnoses. For example, for the Decorator occurrence from Figure 3.4.1, PINOT and SSA classify its State subpattern as a Strategy occurrence, since their criterion for distinguishing State and Strategy fails in this case. 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 the used tools might just identify a tiny "Elemental Design Pattern", EDP (see the definition in [61]). EDPs can serve as 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 occurrence of the Iterator pattern as a witness[2]. Because the tools evaluated in this thesis do not support Iterator, it is approximated by the "ArrayMultiReference" EDP supported by Fujaba.

In general, Strategy, Adapter, Command, State, Template Method and Iterator patterns were preferred since they are more reliable witnesses than EDPs. The reason is that EDPs are smaller and sometimes do not include behavioral aspects. When the tools agree on such witnesses then a much stronger evidence is obtained (in contrast to the case when tools agree on two small EDPs). Thus the returned candidate gets a much higher score.

4.6. Additional Sources of Information

So far only the information provided by the evaluated pattern detection tools was used. However, there are other sources of evidence about design patterns, whose input can be fused too. This information can be used for the following purposes:

- Highlight relevant pattern occurrences when the information about the program structure got from witnesses is not sufficient,
- Distinguish between structurally similar patterns after the witnesses have already been collected.

[2]In Observer, a Subject class uses Iterator to broadcast state changes to observer objects. Composite uses Iterator to run the action on its children.

This section demonstrated the usefulness of class names and program behavior analysis as additional information sources.

Program element names. In the experiments, JTransformer http://sewiki.iai.uni-bonn.de/research/jtransformer/start, a versatile infrastructure for analysis and transformation of Java programs, was used to extract information from variable names and comments. Variable names were intensively used in the experiments to find pattern candidates manually. The names of OBSERVER classes often contain "Listener" or "Observer". The notification method name often starts with the word "fire" - for example, "fireViewChanged". The class names in the Facade and in the Mediator patterns typically contains "Service", "Repository" etc. In the Visitor pattern, the VISITOR classes are typically named as "...Visitor" - for example, $JavaParserVisitor$ from JRefactory 2.6. The information got from class names was especially valuable in spotting implementation variants when the information got from witnesses alone was vague. This approach was especially effective in spotting so-called *1:1 Observers* (an Observer implementation variant where a subject is linked only to one observer that is pointed to by the scalar field owned by a subject). 1:1 Observers were distinguished from structurally and behaviorally similar Strategy or Bridge occurrences by using program element names.

Program behavior. 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 (CoR) and Dynamically Typed Proxy patterns have the same set of witnesses: Command and State. In contrast to Decorator/CoR, in Dynamically Typed Proxy the points-to set of the parent field consists solely of objects whose types are sibling types for the proxy object. To distinguish CoR from Decorator the control flow of the relevant forwarding method(s) is examined to check whether forwarding is done conditionally or not.

4.7. Evaluation of the Data Fusion Approach

The data fusion hypothesis was evaluated manually. Several tens of examples were taken from the repositories listed in Table 3.1.1. Occurrences of the Observer, Visitor, Decorator, Chain of Responsibility, Proxy, Bridge, Mediator and Facade patterns were detected. The data fusion approach delivers better accuracy than each individual tool. Precision was increased as well as recall, since each reconstructed pattern removes a false negative and several false positives (its witnesses) from the output set of the basic pattern detection tools.

The remaining false positives occur because 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.

		Precision				
		FUJABA	PTIDEJ	PINOT	Similarity Scoring	Fusion
Patterns	Decorator (Proxy, COR)	---	32%	60%	24%	77%
	Visitor	30%	25%	50%	26%	73%
	Observer (Composite)	29%	25%	43%	24%	75%
	Bridge	12%	---	14%	---	14%
	Mediator (Façade)	---	---	20%	---	20%

(A) Precisions

		Recall				
		FUJABA	PTIDEJ	PINOT	Similarity Scoring	Fusion
Patterns	Decorator (Proxy, COR)	---	70%	60%	80%	100%
	Visitor	50%	50%	30%	60%	80%
	Observer (Composite)	50%	40%	30%	50%	60%
	Bridge	40%	---	40%	---	40%
	Mediator (Façade)	0		100%		100%

(B) Recalls

FIGURE 4.7.1. The accuracy of the data fusion approach

Full evaluation details are provided in [42, 28]. Fig. 4.7.1 provides the accuracies delivered by several individual tools and by the data fusion approach. The numbers are averaged over the repositories listed in Table 3.1.1.

4.8. Conclusions

Data fusion can benefit from future advances in the detection quality of any of the basic tools, no matter by which techniques they are achieved. In addition, 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. The following sections discuss the ways to advance the data fusion approach.

Improve underlying tools. Obviously, data fusion does not replace work on further improvement of existing tools. The main areas to be addressed are dataflow analyses, support for transitivity and implementation of additional pattern detectors, such as the missing detector for Iterator discussed above. Analyzing of program element names (mentioned also in Tonella et al. [67]) also seem to be promising. Data fusion will be able to take advantage of any improvements of the analysed tools or the availability of new tools.

Support various implementation variants of witness patterns. Witnesses themselves are implemented in a variety of ways. Therefore, it is important that the pattern detection tools that detect witnesses accept the corresponding implementation variants. 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 pattern detection tools that apply static behavioural analyses, PINOT and DP-Miner, support only a few straighforward implementation variants. 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.

Need for 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 and SSA 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, but not for all supported patterns (Table 1). Also, different tools do not assign qualitative labels ($High, Low$) to their scores, making them incomparable. For example, SSA rejects the candidates scored lower than 50%. On the other hand, Decorator candidates scored with 44% should be accepted (the numbers are based on the conversations with the authors).

These observations motivate the need for a commonly agreed meta-model and an uniform exchange format for pattern detection tools to enhance the power of data fusion and to do it automatically. These issues are addressed in Chapter 5.

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

Table 1. Roles reported by the different tools.

CHAPTER 5

A Common Exchange Format for Design Pattern Detection Tools

The previous chapter demonstrated how data fusion improves the overall accuracy. However, automatic data fusion was hindered by diverse output formats of pattern detection tools. This chapter addresses this limitation by developing a common exchange format for pattern detection tools referred to in this thesis as DPDX. DPDX is based on a well-defined and extensible metamodel and facilitates comparison, fusion, visualisation, and validation of the outputs of different DPD tools.

Section 5.1 motivates the need for a common exchange format. Section 5.2 presents the requirements to an exchange format that were distilled from practical evaluations of existing tools and from thorough literature review. Section 5.3 presents the concepts on which the proposed exchange format is built. Section 5.4 describes the metamodel of DPDX. More details of DPDX can be found in [43, 44].

5.1. Motivation

A common exchange format for pattern detection tools would be beneficial to achieve a synergy of many different tools that participate in a federation and interact to produce new value. This federation and the common format is also an invitation to the program comprehension, maintenance, and reengineering research communities to contribute individual tools.[1]

For example, visualisations of tool outputs could be implemented in a separate component that gets relevant information from other components using a the common exchange format. It would be possible to automate the process of collecting, comparing, and evaluating the outputs of different tools, which is currently a manual, error-prone, and time-consuming task. Moreover, public repositories of pattern occurrences[2] would benefit greatly from a common exchange format. These repositories are important in design pattern detection research as a reference for assessing the accuracy of tools [53]. In addition, a common exchange format would also help in achieving an automated round-trip in pattern detection tools (see Albin-Amiot et al.

[1]For an introduction to the used DPD terminology please refer to [42]

[2]See, for instance, PMART (http://www.ptidej.net/downloads/pmart/) and DEEBEE [25].

[1]), including pattern detection, collection, fusion, visualisation, validation, storage, and generation.

5.2. Requirements to an exchange format

The common exchange format encourages reporting of internal information that is available in most tools. It acts as a contract to which tools should conform in order to be a part of the aforementioned federation. A suitable exchange format must fulfill the following requirements:[3]

- **Language independence.** The exchange format must be specified without using language-specific concepts.
- **Completeness.** The format must be able to represent program constituents at every level of role granularity
- **Standard compliance.** The exchange format must conform to some text exchange formats like CSV or XML.
- **Identification of role players.** Each program constituent playing a role must be identified unambiguously.
- **Identification of candidates.** Each candidate must be identified unambiguously and reported only once.
- **Justification.** To ease result assessment, the format must include explanations of results and scores expressing the confidence of a tool in its diagnostics.
- **Comparability.** The format must enable reporting of the motif definitions assumed by a tool and the applied analysis methods to allow other tools to compare results.
- **Reproducibility.** For reproducing the results, the tool and the analysed program must be explicitly reported.
- **Specification.** The exchange format must be specified formally to allow DPD tool developers to implement appropriate generators, parsers, and–or converters.

The output formats of several pattern detection tools and repositories are reviewed in [43, 44]. No individual output format fulfills all the above mentioned requirements.

5.3. DPDX Concepts

This section develops the concepts on which DPDX is based. It is explained how DPDX addresses each of the requirements stated in Section 5.2, overcoming the limitations of existing output formats.

Completeness. For completeness, the output format supports at least reporting each program element that can possibly play a mandatory role. DPDX allows reporting nested and top-level types (interfaces, concrete and abstract classes), fields, methods and individual statements (including field accesses and method invocations). Reporting role mappings at all possible

[3]These requirements have been formed during intensive practical evaluations of existing tools and thorough literature review.

granularity levels improves the presentation of the results and eases their verification by experts and use by other tools.

Standard Compliance. The implementation of DPDX is based on the XML format.

Identification of role players. In DPDX, every program element is identified *unambiguously* by a path in an abstract syntax tree (AST) representation of the respective program. The identification schema is therefore *stable*, i.e. not affected by changes in the source code that are mere formatting issues (e.g. inserting a blank line) or reordering of elements whose order has no semantic meaning.[4]

Identification of candidates. The set of all role assigments identified by a pattern detection tool defines a projection graph (the formal treatment see in Section 7.2) whose nodes are the program elements playing roles and arcs being the relations between these elements (subtyping, method invocation etc). A candidate is the set of nodes in a connected component of the projection graph where edge directions are disregarded.

Justification. Justification of diagnostics consists of confidence scores, reported as real numbers between 0 and 1, and textual explanations. Justification can be added at every level of granularity: for an entire candidate, individual role assignments and individual relation assignments.

Comparability. DPDX supports comparability by specifying a precise metamodel of schemata, enabling tools to report their schemata.

Reproducibility.A result file contains the diagnostics of a particular tool for a particular program. To enable reproducing the results, it must include the name and version of the producing tool and the name, version, and the URI of the analysed program. Names and versions may be arbitrary strings. The URI(s) must reference the root directory(ies) of the analysed program. The URI field is optional, since the analysed program might not be publicly accessible. The other fields are mandatory.

Specification.The common exchange format is specified by a set of extensible metamodels that capture the structural properties of the relevant concepts, e.g., candidates, roles and their relations (Section 5.4). Unlike in the PADL metamodel of Albin-Amiot et al. [2], the possible kinds of program constituents are no first-class elements of the metamodel but are captured by a set of predefined values for certain attributes in the metamodel. This ensures easy extensibility since only the set of values must be extended to capture new relations or language constructs whereas the metamodel and the related exchange format remain stable.

.

[4]This is necessary to compare DPD results across different program versions, when analysing the evolution of pattern implementations over time.

5.4. The architecture of a pattern candidate

A *design pattern schema* is a set of named roles and named relationships between these roles. A *role* has a name, a string indicating of the kind of program element that may legally play that role (e.g. a class, method, etc.), a set of associated properties (e.g. whether role players are abstract) and a specification of the role cardinality, which determines how many elements that play the role may occur within the enclosing entity. Mandatory roles have cardinality greater than zero.

A *relationship* abstracts collaborations between the players of involved roles. It has a name that indicates the meaning (e.g. Inheritance, Control Flow), the mandatory sign (indicating whether can be dropped without regarding a given candidate as false), the transitivity sign (indicating whether it abstracts a transitive relation) and cardinalities specifying how many elements that play a particular role can be related on either relationship end.

A *role mapping* maps roles and relations of the schema to elements of a program so that the target program elements are of the required kind, have the required properties and relationships and fulfill the cardinality constraints stated in the schema.

A *detection result* contains a set of diagnostics produced by a tool for a given program. Each diagnostic contains a set of role and relation mappings and a reference to the pattern schema whose roles and relations are mapped. Each role mapping references a mapped role and the program element that plays the role. A relation mapping references the mapped relation, a program element that serves as relation source and an element that serves as relation target.

Note that this metamodel can accomodate arbitrary languages and the evolution of existing languages without any change in the metamodel because language level concepts are not first class entities of the metamodel but just values assigned to role kinds and relationship names. In order to make different tools understand each other, it is sufficient to agree on a common vocabulary, that is, a set of "kind" values with a fixed meaning. For instance, the "kind" *class* generally represents an object type and the distinction between interfaces, abstract classes and concrete classes is represented by the property, "abstractness", with predefined values $interface$, *abstract* and *concrete*.

Kniesel et al. [**43, 44**] define the above mentioned metamodels formally and outline its implementation in XML. The implementation is also provided in the Ph.D.thesis of Fülöp [**24**].

Part 3

Approach 2: Fusion of Techniques

CHAPTER 6

An Object Level View of Design Patterns

The research community needs a commonly agreed, domain-independent platform to reason about design patterns. This platform must provide a common basis so that researchers can express formally why a particular candidate is indeed an occurrence of a certain pattern.

The approach taken in this thesis is to reason about the *interactions* within pattern occurrences at the object level, in contrast to the approaches presented in prior pattern-relevant work, which focuses on the structural description of designs rather than on the conceptual object interaction expressed by these designs. It is worth noting that description of object level interactions has been part of design pattern descriptions from the very beginning [**27**], which emphasizes its relevance for expressing the essence of a pattern.

This section elaborates this idea, arriving at pattern descriptions that are abstracted from concrete program structure. Thus, many design variants that implement the same conceptual collaboration can be easily captured. This is the key to achieving a high accuracy since the behavioral characteristics of design patterns can be stated more clearly at the object level.

To identify, define, and organise the object interaction characteristics of the design patterns addressed in this thesis, the following activities were performed:

- Descriptions provided by Gamma et al. [**27**] and in other related literature were studied.
- Experimental work, such as the one of Wegrzynowicz et al. [**73**] (addressing creational patterns) and of Riehle [**57, 18**] was reviewed.
- Own experimental studies of the occurrence of collaborational patterns in existing software and of their detection by available tools were carried out (see [**42**]).

All activities were performed in close collaboration with researchers that are active in the design pattern detection field (see Acknowledgement section).

This chapter

- describes pattern-relevant concepts at the object level (Section 6.1),
- discusses pattern-relevant collaborations between entities at the object level - Section 6.2,

- and finally, compares the program level and the object-level viewpoints (Section 6.3).

6.1. Pattern occurrences at the object level

The following entities available at runtime are needed to reason about a a pattern occurrence:

- Memory locations: objects and their fields, local variables and shared variables.
- Events: method invocations on receiver objects.

The type of object is identified by its class with the help of Run-Time Type Identification system (RTTI). For a method, it is known whether it is a construction method for a certain object.

Method invocations play a central role in all patterns. They can be derived from method call sequences. In some situations , one needs to know whether a certain method invokes some methods either directly or indirectly so intermediate methods are taken into account. The order of intermediate method calls and the number of invocations of a certain method on a given object o are irrelevant.

Dataflow and points-to sets are computed by examining the contents of the corresponding memory locations. Dataflow is used, for example, in a Dynamically Typed Proxy occurrence (Section 3.4) to verify whether the parent field points to an object whose type is a sibling of the proxy type.

A *pattern occurrence* is seen as a set of assignments of certain roles to the runtime entities. For example, in an Observer pattern occurrence, each of the roles Subject State and Observer State is played by at least one variable.

The core constraints for a pattern P restrict interactions between the role players in an occurrence of P, selecting only the role players that interact in a certain "pattern-specific" way. A core constraint for a pattern P serves as the necessary condition fulfilled by participants of the occurrences of P (except for rare exotic occurrences). Section 6.2 discusses the core constraints. The scenarios from the section "Reasons behind false positives" in Section 3.4 are examples of using of these constraints.

6.2. The Core Constraints

Exclusive Data Flow Source / Target. In the design patterns like CoR, Decorator, Proxy, Bridge, State and Strategy caller method(s) must be invoked on a single dependent object. In the Tree Forwarder pattern the forwarding call(s) must be addressed to more than one parent object. In general, the set of dependent objects participating in pattern-relevant interactions is restricted. Section 3.4 provides the examples of forwarding to a single object in the Decorator pattern and of forwarding to multiple objects in the Tree Forwarder pattern.

Denote the relevant callee method invocations by $mc_1...mc_n$. The Exclusive Data Flow Source constraint captures technically the above mentioned idea, requiring all dataflow into the receiver variables of $mc_1...mc_n$ originate at a certain object(s) $o_1...o_m$.

In the Singleton pattern, the maximal number of Singleton objects must be known in advance. The Exclusive Data Flow Target constraint enforces this condition, requiring all singleton objects flow into certain shared variable(s).

Same Operation. The crucial point of forwarding-based design patterns (Decorator, Proxy, Chain of Responsibility, Composite) is that the caller method implementing a particular operation in a master object invokes the *conceptually same* operation on a dependent object. In Observer, State, Strategy, Bridge a caller method and the invoked callee method(s) must represent different operations. So the question arises, how to capture technically the idea of "conceptually same" operations.

Fortunately, all object-oriented languages, which are the focus of this thesis, answer this question in the same way. They typically distinguish a conceptual *operation* from the *methods* that are its various implementations. The bottom line is that at run-time methods for the same operation have the same position in the virtual function table (*vtable*) of objects of different concrete types that are specializations of a common supertype (see [65, 69] for more details). This principle is the same in prototype and class based languages, the only difference being that each prototype is its own concrete type and thus has its own vtable, whereas all instances of a class share the vtable of the class.

Thus the requirement that the implementation of a particular operation in an object invokes the conceptually *same* operation on another object can be rephrased to: "A method with index i in type T_1 calls a method with index i in an object of type T_2 and the two types, T_1 and T_2, have a common supertype T_{CS} that also has index i in its vtable." This condition implies that the methods with index i in T_1, T_2 and T_{CS} implement the same operation. The condition can be checked statically and at run-time since the index of invoked operations is determined statically by the compiler and is part of the executing code.[1]

For example, in a Decorator occurrence let D be a decorator object by and D_{next} be an object pointed to by the parent field of D. The forwarder methods $D.M$ and $D_{next}.M_1$ respectively represent the same operation. In contrast, in an Observer occurrence a notifier method N owned by a subject S and an update method U owned by an observer object O that is related to S must represent different operations.

[1]Unless agressive optimization techniques are applied.

Update Field. In an occurrence of a decoupler pattern (e.g. Observer, Composite, Mediator, State, Strategy and Bridge, see also Sec. 2.4) the field F owned by a master object O can be modified after O is created.

The "Update Field" constraint captures technically the just mentioned idea, requiring that some non-construction method M owned by O forms the scope of the dataflow into F[2] and M is called by some *external client method* that is not owned by O.

Note that the Update Field constraint in the context of the Observer / Composite patterns (when the fields *observers* or *children* stand for F), if violated, **does not lead to the rejection of the corresponding pattern candidates.** Nevertheless, their scores are decreased. This conclusion was made after observing several (rare) Observer / Composite occurrences where the Update Field constraint is violated since pattern-relevant fields were modified only within the construction methods of their owners. No State/Strategy/Bridge occurrences with violated Update Field constraint were found. The Update Field constraint is said to be *mandatory* in the Observer / Composite patterns and *optional* in the State/Strategy/Bridge patterns.

State Propagation. Assume that in an Observer occurrence the field $Source$ (respectively, $Target$) holds the state of a subject S (respectively, of an observer O). The State Propagation constraint guarantees state change propagation by requiring that a dataflow from $Source$ to $Target$ exists.

Note that $Source$ is not necessarily owned by S and $Target$ is not necessarily owned by O. For example, $sstate_1 ... sstate_k$ might be fields of an object pointed to by a field F owned by S. In this case $Source$ is said to be *transitively referenced* from S and $Target$ is said to be *transitively referenced* from O.

The State Propagation constraint is optional. Assume that a notifier method invokes an update method $O.U(...)$. According to the program logic O can consider an invocation of U as a certain "state change signal" without checking the dataflow between corresponding state variables. More details are provided in Section 7.3.

Control & Data Flow Order. In several collaborational patterns (Memento, State, Observer) the accesses and updates of a certain pattern-relevant field occur in a certain order. Assume that in a such an occurrence a method M accesses a field f and a method U updates f. All invocations of M and U are within the control flow of a method N. Ordering can be done in two ways:

Sequential Order: All invocations of M (if any exist) are performed *before* the invocations of U. In terms of regular expressions, an

[2]No matter whether F is collection-typed (Observer, Composite) or scalar-typed (Bridge, State, Strategy) - both implementations are captured.

execution of N must satisfy the condition M^*U^+.[3] For example, in an Observer occurrence a method N stands for a notifier method, M stands for a method accessing a subject state field f and U stands for a method updating f.

Interleaving Order: A non-empty set of invocations of M and an invocation of U interleave; in addition, an invocation of U precedes the whole execution. In the terms of the regular expressions, an execution of N must satisfy the condition $(UM^+)^+$. For example, in a State occurrence a method N stands for an Action method, M stands for a method that accesses a current state field f in order to execute a state-related action and U stands for a method updating f.

6.2.1. Sibling Creation. In Dynamically Typed Proxy (see Section 3.4) occurrences the type of a master object (in this case, proxy object) O and of the dependent object o (i.e. real subject) to which O forwards are siblings types. In contrast, in Decorator and CoR occurrences O and P may belong to any type.

The Sibling Creation constraint captures the just mentioned idea technically by restricting the points-to set of the dependent object field F owned by a master object O. That is, $pointsTo(O.F)$ consists only of objects whose types are siblings of $type(O)$.

Restrictions on Object Coverage. Assume that a caller method M invokes a callee method m on objects pointed to by a set of relevant fields $f_1..f_n$. The corresponding method calls are denoted by $mc_1...mc_k$. In certain patterns, the following restrictions on invocations of m occur:

- *Full Coverage* - m is invoked *on each object* o pointed to by $f_1....f_n$ *in each execution* of M.
  - This rule applies to occurrences of Decorator (a decorator object always forwards to its parent object), Observer (a subject always propagates state changes to all its observers) and Composite (a container always propagates an operation to all its children).
- *Partial Coverage* - m *is not* invoked on *some* object o pointed to by $f_1...f_n$ in *some* execution of M (this execution is called *a diverting execution*) but *is* invoked on o in *other* execution(s). In other words, a) in some execution of M no call of m is executed on some object $o \in pointsTo(f_{i\in\{1..n\}})$ and b) at least one execution of M contains for each $o \in \bigcup_{1\leq j\leq n} pointsTo(f_j)$ a method call mc_i executed on o.

  - This rule applies to occurrences of the Chain of Responsibility pattern (a handler object forwards to its next object only in

[3]The usual notation for regular expressions is used. That is, $*$ denotes a non-negative number of occurrences and $+$ denotes a positive number of occurrences.

some execution) and to false Observer / Composite candidates where not all dependent objects are addressed - for example, due to loop breaks.

Note that diverting executions in the Partial Coverage constraint must be formed by tests that reflect application logic. For example, if a method call $f.callee()$ resides within a branch of an if-statement $if(f! = null)$ then the path corresponding to the else-branch does not lead to a diverting execution. This is else-branches of such null checks contain "cleanup" actions that are done by attempt to perform an invocation of the null pointer.

Unique Created Product Type. In any occurrence of a creational pattern, any two creator method invocations that refer to different creator methods, or carry different parameters, or both, must yield product objects of different types. This idea is summarized into the the Uniquely Created Product Type constraint.

6.3. The object level vs. the program element level

This section discusses how the object-level pattern concepts from Section 6.1 are reflected in the reviewed design-level pattern detection approaches. If some concept is used in a limited way, reasons behind false positives / negatives are discussed.

Most tools addressed in this thesis (SSA [68], SPQR [36], D-Cubed [73], PINOT [58], DP-Miner [16], Columbus [20]) employ static analyses. Ptidej [50, 29], Fujaba [71] and DPRE [47] additionally analyze selected execution traces to improve the precision. Columbus [20] applies machine learning techniques in addition to static analyses.

Object-Related Issues.

Specifying objects and their parts. Pattern specifications used in the tools based on static analyses *prescribe certain distribution of functionality* between the class(es) that form the types of pattern-relevant objects. For example, in the Observer pattern the observers variable and the notifier method(s) should be owned by the Subject class; the state variables should be declared in the Concrete Subject class(es).

SSA [68], Ptidej [50], SPQR [36], Fujaba [71] and D-Cubed [73] support the limited case of the subtype transitivity where the number of intermediate classes is limited. They declare a class C_S as a mandatory role player and its subclass C as a player of an optional role. Thus it is possible to accept candidates in which certain fields / methods are owned by C_s and the remaining ones are owned by C. Still, certain variables / methods are required to coexist in the same class, which might not actually be true. For example, the observer variable and the notification method are still required to be owned by the (same) Subject class.

PINOT [58] and DP-Miner [16] impose additional constraints for improving precision; for example, a certain class (Subject) must be abstract. However, this decreases recall.

The conclusion is that a solution that accepts *all possible distributions* of functionality among the classes that contribute to the implementation of a particular object is needed.

Object sets. The Total / Partial Coverage constraints (Section 6.2.1) involve the notion of object set (a method must be executed on all or on some objects *belonging to a certain set*). The tools based on static analyses model the "object set" concept via a collection- or array-typed variable. Iterations are expressed as loops.

Such specifications cover only a subset of valid occurrences. If the maximal number of referenced objects is known in advance, more efficient implementations are possible. For example, there exist implementations of Observer, Composite and Visitor that use a fixed number of scalar-typed variables that point to dependent objects. An iteration over a set of dependent objects is implemented via a sequential operator.

The tools based on dynamic analyses (Ptidej and Fujaba) do not fall into this trap since they focus on sequences of run-time events, independent of the structure of the program that produces them.

For a better static approach object sets must be abstracted in the specification of a pattern, which must not depend on implementation details such as arrays, collections or other ways to implement sets.

Method-Related Issues.

Same Operation. All tools reviewed in Chapter 3 support the "same operation" concept by checking whether M_2 overrides M_1 or not. To detect Observer candidates they apply a more liberal check: whether a notification method and update methods reside in different subtype hierarchies. Such a check misses some Observer occurrences mentioned by Riehle [18, 57] where a notification and update methods reside in the same subtype hierarchy but have different signatures.

Specifying method invocations. Certain tools based on static analyses (PINOT [58] and DP-Miner [16]) represent a method invocation by a method call statement. SSA [68] and D-Cubed [64] support the method invocation relation, i.e. method pairs (F, G) when a method F calls a method G. In all these approaches the information about the receiver object is not represented, making it impossible to express several patterns, for example, Dynamically Typed Proxy.

Ptidej [29], SPQR [36] and Fujaba [71] overcome this limitation by supporting the "method delegation" entity. A method delegation is created for each triple $<Caller, Callee, V>$where a method $Caller$ might invoke a method $Callee$ and V stores the receiver object.

Method invocation transitivity. The method invocation relation supported by PINOT [58], Ptidej [29], Fujaba [71], DP-Miner [16], D-Cubed [64] and

SSA [68] is not transitive, i.e. intermediate methods are not allowed. The partially transitive method invocation relation (the number of intermediate methods is limited) is supported by SPQR [36].

Conditional method invocations. Except for PINOT [58], the method invocation relations mentioned in the previous sections do not include the information whether a caller method runs a callee method in *every* program execution. Thus some CoR occurrences might be misrecognized as Decorators. PINOT [58] builds an extended variant of the method invocation relation, enabling to figure out whether method invocations are performed within a branch of a conditional operator.

Variables and data flow.

Variables. The tools based on static analyses represent variables as field declarations owned by classes and local variable declarations owned by methods.

SSA [68], Columbus [20], Ptidej [29], D-Cubed [73], SPQR [36] and DPRE [47] do not specify method-related variables, arguing that pattern specifications at the granularity level of local variables and parameters are too restrictive. Stencel et al. [64] justify this decision by showing that some Factory Method implementation variants that require a factory method to have no parameters do not cover all possible Factory Method implementations.

Data Flow. SSA [68], Ptidej [29], DP-Miner [16], SPQR [36] and Fujaba [71]) do not support dataflow analyses. As a result, they do not distinguish between Statically Typed and Dynamically Typed Proxy implementations (Section 3.4).

Limited dataflow analyses are implemented in D-Cubed [73] and PINOT [58]. These are intended to ensure that all Singleton objects are stored in a finite number of static variables. But these approaches are limited in their capabilities. They require callee methods to be invoked directly on some pattern-relevant field F, failing to detect invocations on a local variable into which objects flow from F.

In all tools reviewed in this thesis method invocations that trigger accesses to and updates of a certain field are not ordered; that is, restrictions discussed in Section 3.4 cannot be verified.

Grouping of elements in pattern candidates. Except for SSA [68], all tools reviewed in this thesis represent a pattern candidate as a set of program entities fulfilling certain roles. If more than one program element plays a role, multiple candidates are reported. That is, all role cardinalities are [1..1] be default. For example, if a Subject class contains two notification methods, PINOT reports two Observer candidates, one for each notification method. SSA [68] groups several method role players around the owning classes, thus supporting the cardinality $[1..\infty]$ for method role players.

In the approach of Kim et al. [41] cardinalities are specified and enforced for roles of each kind.

6.4. Conclusion

In the static-analysis-based tools design patterns are originally specified in terms of program elements which is either too restrictive, covering only a subset of possible implementation variants and producing false negatives or too liberal, creating false positives.

The object-level specifications are used by the pattern detection tools employing dynamic analyses and are checked against selected program execution traces. This is done in order to filter out false positives emerging because of too liberal static behavioral checks. Dynamic analysis has proven to improve detection quality if sufficiently good test coverage can be achieved. This is, however, a time and space-consuming-task. See [**51**] for more details.

As an alternative, Columbus uses machine learning. It learns the typical range of metric values for program elements participating in valid pattern occurrences, thus discarding some false positives. However, keeping training sets up-to-date might be unfeasible, especially for quickly changing software systems that contain new pattern implementations.

This chapter shows that the object-level descriptions are better suited for pattern specifications, unifying cases that appear to be different implementations in specifications that commit prematurely to the program element level. This thesis advocates the following approach:

(1) First specify patterns in generic terms that are independent of a concrete program analyses (static or dynamic). See Chapter 7 for the details.
(2) Then map the obtained generic specifications into the concrete level. Chapter 8 maps the obtained generic specifications into the program element level, taking care to create sufficiently general abstractions for different implementation variants of the same object-level concept.

Such an approach avoids the discussed pitfalls and makes it easier to compare static and dynamic approaches since it starts from generic descriptions.

CHAPTER 7

Specifying Pattern-Specific Interactions

Following the informal discussions in Chapter 6, this section specifies formally the interactions characteristic for the patterns treated in this thesis. Each pattern P is associated with its *interaction specification*. An interaction specification consists of a set of *roles* that are played by elements of a software system. This is formally specified in Section 7.1. Interactions between players of two different roles are abstracted into a *relationship* between the roles (Section 7.2). *Core constraints*, motivated in Section 6.2, restrict the collaborations between role players in a certain "pattern-specific" way. Section 7.3 describes the core constraints formally as predicates written in a first-order-logic language. Section 7.4 specifies the patterns that are implemented in the tool of the Decorator pattern. Section 7.5 summarizes this chapter.

Roles, relationships, constraints and the software system abstraction form the framework for building interaction specifications. This framework is referred to throughout the thesis as *Pattern Interaction Framework*. It is used in Section 7.4 to specify the characteristics of collaborational design patterns (Sec.2.4).

7.1. The Software System Abstraction

The term "software system" is used as a common abstraction for programs and execution traces. Thus interaction specifications based on the software system abstraction, can be easily mapped to dynamic analyses (object level) and to static analyses (program element level). At the object level, the interaction specification for a pattern P covers the trace(s) of a program implementing P. At the program element level, the interaction specification for P covers all implementation variants of P that express the same behaviour.

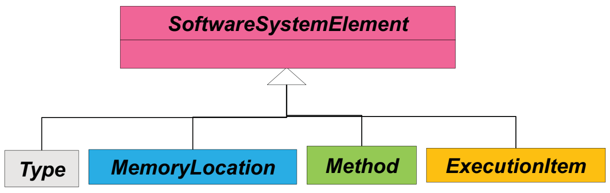
Software System Metamodel. The model of a software system is represented in Fig. 7.1.1. The classes represent categories of software system elements; the associations represent relations and functions specified in Section 7.1. A software system consists of elements from the following sets:

- **Types** are denoted by the set *Types* and represented by the class *Type* in Fig. 7.1.1b.

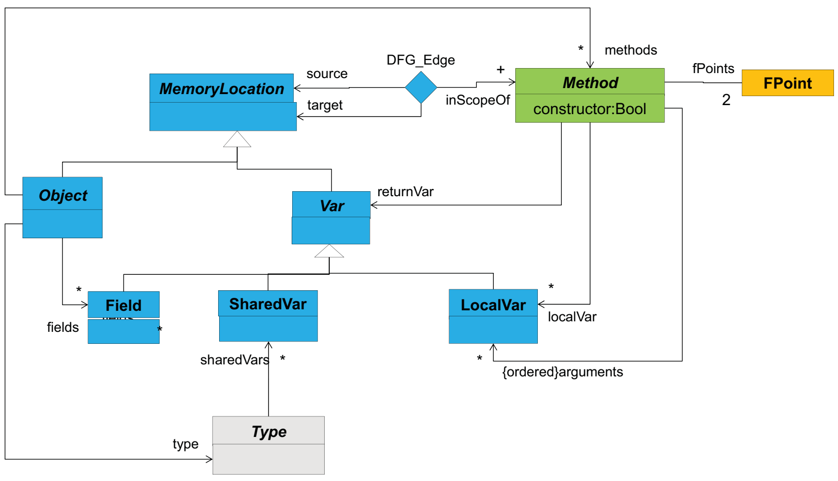
- **Memory locations**, denoted by the set $MemoryLocs$, are a common abstraction for objects and variables, i.e. $MemoryLocs = Objects \cup Variables$.
  - **Objects** are denoted by the set $Objects$ and represented by the class $Object$ in Fig. 7.1.1b.
  - **Variables** are denoted by the set $Variables$ and represented by the class Var in Fig. 7.1.1b. They serve as a common abstraction for fields, shared variables, local variables or method parameters:
    - Fields are denoted by the set $Fields$ and represented by the class $Field$ in Fig. 7.1.1b.
    - Shared variables are denoted by the set $SharedVars$ and represented by the class $SharedVar$ in Fig. 7.1.1b.
    - Local variables and parameters are denoted by the set $LocalVars$, represented by the class $LocalVar$ in Fig. 7.1.1b.
- **Methods** are represented by the class $Method$ in Fig. 7.1.1b. Methods have a *constructor* attribute with default value $FALSE$. Construction method are indicated by the attribute value $TRUE$.
- **Execution items** are denoted by the set $ExecutionItems$ and represented by the classes in Fig. 7.1.1c). An execution item is represented by the abstract class $ExecutionItem$, which abstracts elements that participate in pattern-relevant control flow:
  - *method calls* are denoted by the set $MethodCalls$, represented by the class $MethodCall$,
  - *method start / method exit* points are elements of the sets $Starts$ and $Exits$ respectively. They are represented by the class $FPoint$ whose attribute *kind* can take on the values *entry* or *exit*. The start and the exit point for a method M is denoted by $Start_M$ and End_M respectively. They wrap the execution of M.
  - *null checks* are elements of the set $NullChecks$, represented by the class $NullCheck$. Null checks represent checks whether a variable has the special value *null*. They are associated with the variable whose content is examined.

The set $SoftwareSystemElements$ is the union of the above sets. It is represented in the model by the abstract class $SoftwareSystemElement$.

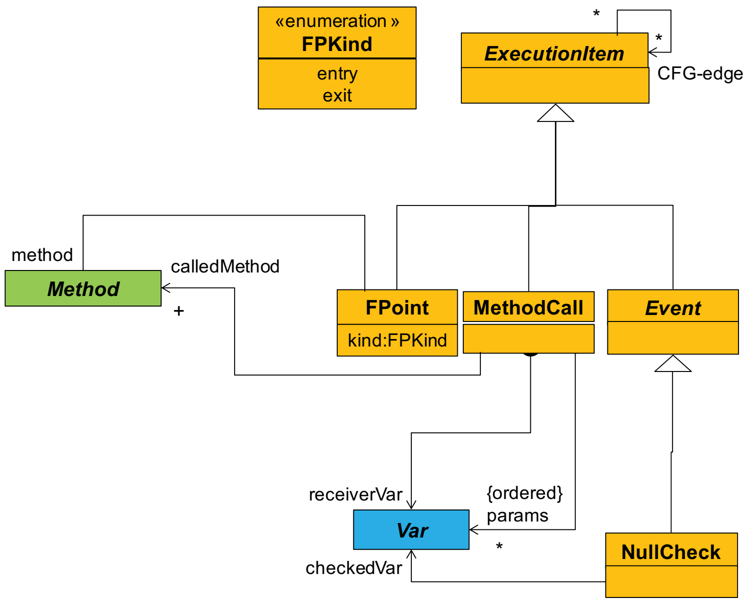
Relations and functions. The collaborations between software system elements are expressed using *relations* and *functions* on the software system elements (Section 7.1). In the software system model (Fig. 7.1.1) relations are modeled as associations. Functions are modeled as attributes or many-to-one associations. The relations and functions do not involve details that are specific for the object level or for the program element level. Thus



(A) The top level elements.



(B) Types and memory locations



(C) Execution items

FIGURE 7.1.1. The model of a software system

they serve as an abstraction layer, shielding the generic design pattern specifications from the concrete program analyses employed by a design pattern detection tool.

Structural Relations and Functions. The structural relations and functions are presented on the software system metamodel excerpt in Fig. 7.1.1b. The relations and functions standing for ownership, type information, construction methods and method call decompositions are basic. The remaining relationships and functions are derived from the basic ones; the derivation steps are outlined.

Ownership. The relation $field \subseteq Objects \times Fields$ (resp. $method \subseteq Objects{\times}Methods$ associates objects with their fields (respectively, methods) and is modeled using the $fields$ association between the UML classes $Object$ and $Field$ in Fig. 7.1.1b. For example, the expression $field(O, F)$ means that the object O owns the field F.

Local and method argument variables are defined by the relationship $localVar \subseteq Methods{\times}LocalVars$ and by the function $argument : Methods\times Integers \rightarrow LocalVars$. In the latter case, an integer number specifies the parameter position within the method's signature. Each method M that returns a value owns a pseudo-variable $return_M$ that points to the value returned by M. When a method returns a value via a statement $v = M()$ a data flow from $return_M$ into v is triggered. Local and method argument variables are modeled using the associations $localVars$ and $arguments$ in Fig. 7.1.1b.

If an instance method / variable M and another instance method / variable N are owned by the same object then M and N are said to be *siblings*.

The relation $sharedVar \subseteq Types \times SharedVars$ is defined as well to describe shared variables (also known as "class variables" in Smalltalk or "static variables" in Java and C++). Writing $shared(T, Var)$ means that an object type T is a type that owns the variable Var whose value is shared by all the instances of T. Shared variables are modeled using the associations $localVars$ and $arguments$ in Fig. 7.1.1b.

Type Information. The function $type : Objects \rightarrow Types$ returns the type of an object. It is modeled using the association $type$ in Fig. 7.1.1b.

Construction method. The function $isConstructor : Methods \rightarrow Boolean$ returns $TRUE$ if a given method is a construction one; otherwise it returns $FALSE$. It is modeled using the attribute $Method.constructor$ in Fig. 7.1.1b.

Method Call Parts. The functions $receiver : MethodCalls \rightarrow Variables$,$param : MethodCalls \times Integers \rightarrow Variables$ and the relation $calledMethod \subseteq MethodCalls{\times}Methods$ provide access to the three relevant parts of method calls. For the method call mc,

- $receiver(mc) = v$ means that v is the receiver variable of mc (modeled by the association $receiverVar$ in Fig. 7.1.1c).

- $param(mc, n) = v$ means that the variable v is the n-th parameter passed to mc (modeled by the association *params* in Fig. 7.1.1c).
- $calledMethod(mc, M)$ means that mc invokes the method M (modeled by the association *calledMethod* in Fig. 7.1.1c).

Note that *calledMethod* is a relation: The same call can invoke different methods, depending on the type of its receiver objects.

Sibling Fields and Methods. The relation $siblings \subseteq (Methods \cup Fields) \times (Methods \cup Fields)$ describes sibling fields or methods, i.e. the fields and methods belonging to the same object.

Let the relation $instanceMember(O, Member)$ describe instance members of the object O, i.e.

$instanceMember(O, M) \equiv field(O, M) \vee method(O, M)$.

Writing $siblings(S_1, S_2)$ means that the method / field S_1 and the method / field S_2 belong to the same object O.

$$\begin{array}{c} siblings(S_1, S_2) \equiv \exists O \cdot \\ (instanceMember(O, S_1) \vee instanceMember(O, S_1)) \\ \wedge \\ (instanceMember(O, S_2) \vee instanceMember(O, S_2)) \end{array}$$

External calls. The central requirement in the decoupler patterns (Par. 2.4) is that a relevant field F is updated in the control flow of an *external client* (i.e. that a method that is not a sibling of F) after the owner of F is created. The relation $ExternallyCalled(M)$ checks whether the method M can be called by a method that is not a sibling of M.

At the object level, it is checked that there exists a method E that is not a sibling of M and invokes M.

The situation at the program element level is slightly different. In an incomplete program (e.g. a library) E might be contained in a potentially absent client code. Therefore M is required to be *public*. That is, M can be *potentially* called by some external client.

Behavioural Relations. The relations and functions standing for control flow and data flow graphs are the basic ones. The remaining relationships and functions are derived from the basic ones; the derivation steps are outlined.

Control Flow Graph. The control flow of the method M is defined as follows: $cflow_M \subseteq ExecutionItems \times ExecutionItems$. It can be seen as a graph (referred to in the thesis as *abstracted control flow graph of* M) and denoted by $CFG_{abstr}(M)$[1]. Its nodes are the elements of $ExecutionItems$. An edge $(s, t) \in ExecutionItems \times ExecutionItems$ means that t immediately follows s in $CFG_{abstr}(M)$. Note that in the "immediately follows" relation only the elements belonging to $ExecutionItems$ are considered, i.e. t is assumed to immediately follow s even if certain events are not relevant for pattern logic (for example, individual variable assignments) occur between s and t.

[1]At runtime, this graph is a total order.

A control flow graph edge is described on the software system model excerpt in Fig. 7.1.1c using the association CFG_edge on the class $ExecutionItem$.

Data Flow Graph. The dataflow within the method M is modeled by the relation $dflow_M \subseteq Objects \cup Variables \times Variables$. Writing $dflow_M(Source, TargetVar)$ means that the dataflow from the object or variable $Source$ to the variable $TargetVar$ occurs in the scope of the method M (See the basics chapter , Ch. 2). The dataflow relation of M can be seen as a graph DFG_M.

A data flow edge is described on the software system model excerpt in Fig. 7.1.1b using the ternary association DFG_edge.

Control Flow Reachability. The *method calls reachability* relation $cfreach(mc_1, mc_2, M)$ means that there is a path from the method call mc_1 to the method call mc_2 in $CFG_{abstr}(M)$. The *control flow reachability* relation $cfreach(M, mc)$ means that the method call mc is control flow reachable from $Start_M$. Both relations are computed by traversing $CFG_{abstr}(M)$.

Enumerating paths in the control flow graphs. First some auxiliary definitions are presented. Given the method M and the path $Path$ from $Start_M$ to End_M in $CFG_{abstr}(M)$. Such a path is said to be *through the control flow of* M. Let S be the set of all execution items belonging to $Path$ except $Start_M$ and End_M. Put another way, S makes the order of execution items irrelevant. It is said that S *abstracts* $Path$.

Given the method M, the relation $cfpath : Methods \times P(MethodCalls)$ collects paths without cycles through $CFG_{abstr}(M)$. It is written $cfpath(M, P)$ meaning that the set P abstracts a path through the control flow of M. Conceptually, this relation is computed by traversing each individual pattern relevant *path* in $CFG_{abstr}(M)$ that start at $Start_M$ and ends at $Exit_M$. Pattern-relevant paths represent application logic but not exception flow. More on this topic see in Section 2.5.

Data Flow reachability. The dataflow reachability relation $dfreach(Source, Target, M)$ is computed by finding a path from the variable or object $Source$ to the variable $Target$ in DFG_M. The short notation $Source \rightarrowtail_M Target$ is also used. If the method forming the dataflow scope is irrelevant, the global dataflow reachability relation is defined:

$$dfreach(Source, Target) \equiv \exists M \cdot dfreach(Source, Target, M)$$

The notation $Source \rightarrowtail Target$ is equivalent to $dfreach(Source, Target)$.

Points-to sets. To describe points-to sets, the function $pointsTo : Methods \times Variables \to P(Objects)$ is defined. Writing $pointsTo(M, Var) = \{Obj_1 ... Obj_n\}$ means that the variable Var may point to the objects $Obj_1 ... Obj_n$ and the dataflow is in the scope of method M. . If the dataflow scope is irrelevant, $pointsTo(Var) \equiv \exists M \cdot pointsTo(M, Var)$.

At the object level, the points-to set of the variable Var is computed by monitoring the contents of the memory location corresponding to Var. At the program element level, $pointsTo(M, Var)$ consists of all objects Obj such that $Obj \rightarrowtail_M Var$.

Transitive reference. This section formalizes the transitive reference concept from Section 6.2. The object O, the variable V and the method M are given as the output. Writing $transRef(O, Var, M)$ means that Var is transitively referenced from O. That is, Var satisfies one of the following conditions:

- Var is a field owned by O
- Var is a field of an object pointed to by some field $O.f$.

The dataflow is in the scope of M .

Formally,

$$
\begin{aligned}
&transRef(O, Var, M) \equiv \\
&\qquad\qquad field(O, Var) \vee \\
&\qquad \exists f \cdot field(O, f) \wedge \exists o \in pointsTo(f) \cdot field(o, Var)
\end{aligned}
$$

Enumerating individual paths in data flow graphs. The relation $dfpath : Methods \times Variables \cup Values \times Variables \times P(Variables \cup Values)$ presents individual paths in dataflow graphs. Writing $dfpath(M, Source, Target, P)$ means that the following conditions are met:

- There is a dataflow in the scope of the method M that starts at the variable / object $Source$ and ends at the variable $Target$
- A set P contains all intermediate variables that do not belong to cycles.

The short notation $Source \rightarrowtail_{M,P} Target$ is also equivalent to $dfpath(M, Source, Target, P)$.

Individual paths in DFG_M are computed in two stages:

(1) Build the strongly connected components (SCCs) for DFG_M and collapse these into single nodes, getting the acyclic graph (denoted by $ADFG_M$). The nodes of $ADFG_M$ that do not represent collapsed SCCs describe pattern-relevant variables[2].
(2) Compute all paths from $Source$ to $Target$, traversing each path from $Source$ to $Target$ in $ADFG_M$ and collecting all intermediate nodes that do not represent cycles.

[2]According to the empirical observations, variables that reside on dataflow graph cycles are not pattern-relevant because the relevant variables in the pattern implementation reviewed by the author do not participate in cyclic assignments.

7.2. Roles and Relationships

The *schema* for a design pattern P is a set of roles and relationships between these roles that reflect interactions in an occurrence of P. The core constraints restrict the role players in the pattern-specific way. The subsections of this section specify roles and relationships, assignments of roles to software system elements and provide example schemas.

Roles and Relationships. *A role* is characterized by the name, the cardinality and the role kind. For example, in the Observer pattern the NOTIFICATION METHOD role is played by at least one method that notifies observers of subject state changes.

The role cardinality notation is adopted from the UML. The specified cardinality determines how many elements may or must play the role. Roles with cardinality greater than zero are called *mandatory*. Roles with cardinality including zero are called *optional*. The *role kind* indicates the kind of software system element(s) that may legally play that role (e.g. a class, method, etc.) and is represented by the corresponding subclass of the abstract class *Role* from the metamodel in Figure 7.2.1b. The following role kinds are distinguished, according to the kind of their players:

- *ObjectRole* (objects),
- *TypeRole* (types),
- *VariableRole* (variables),
- *MethodRole* (methods) and
- *MethodCallRole* (method calls).

The class *MemoryLocRole* represents the roles that are played by objects or variables.

Relationships between roles abstract the interactions between the players of these roles that are described by basic relations from the previous section. A *relationship* is characterized by its name and cardinalities for each related role. The relationship name determines the relationship meaning (e.g. data flow). The cardinality specifies the number of players of the target role associated to the number of players of the source role. In the schemata metamodel (Figure 7.2.1a) relationships are represented by the corresponding subclasses of the *Relationship* class:

- *Owns*(*Owner*, *Part*) abstracts the ownership relation. It relates a type, an object or a method role in the first argument to a variable or a method role in the second argument. For instance, an object owns fields and methods, a method owns local variables, a type owns shared variables. The cardinality is one-to-many (e.g. an object owns several fields, a field is owned only by one object).
- *Receiver*(*MethodCallRole*, *VarRole*) abstracts the receiver variable relation. It relates a method call role in the first argument to a variable role in the second argument. The cardinality is many-to-one (several method calls can have the same receiver variable).

- $CalledMethod(MethodCallRole, MethodRole)$ abstracts the called method relation. It relates a method call role in the first argument to a method role in the second argument. The cardinality is one-to-many (a given method call refers to at least one method).
- $CFReach(MethodRole, MethodCallRole)$ abstracts the control flow reachability relation. It relates a method role in the first argument to a method call role in the second argument. The cardinality is many-to-many (a given method can invoke more than one method calls; a given method call can be invoked by more than one method).
- $DFReach(MemoryLocRole, MemoryLocRole)$ abstracts the data flow reachability relation $dfreach(Source, Target)$. It relates a memory location role (object or variable) in the first argument to a variable role in the second argument. The cardinality is many-to-many (a given variable can be reachable from more than one object or variable; more than one variable can be dataflow reachable from a given object or variable).

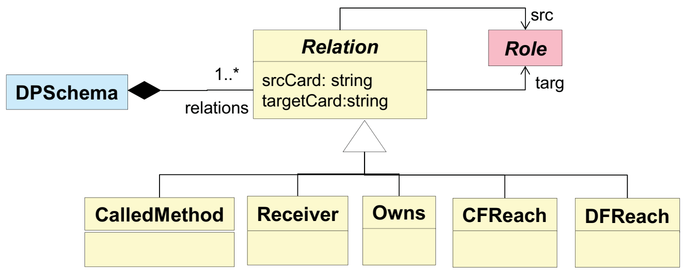
Role assignments and candidates. A *role assignment* maps roles and relations of the schema to software system elements so that the software system elements are of the required role kind, participate in the required relations and fulfill the cardinality constraints and the core constraints specified for a given pattern.

For example, the method calls that play the target role in the $CFReach$ relationship are control flow reachable from the corresponding methods playing the source role. In the Observer pattern, the Total Coverage constraint is applied as follows: the update method call that is invoked by a given notifier method N must be executed in every execution of N on each observer object related to the subject owning N. More information on application of the core constraints on the appropriate software entities see in Section 7.4.

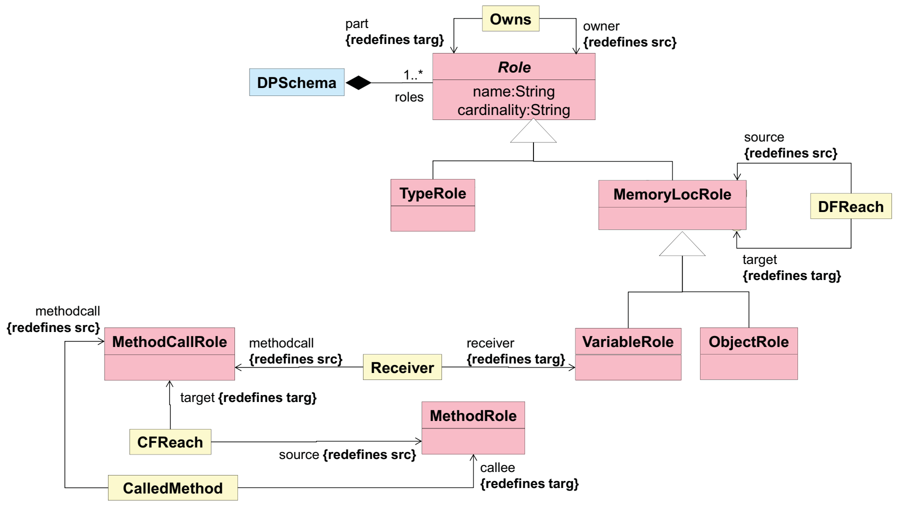
Role assignments are represented by the relation
$plays \subseteq Roles \times SoftwareAbstractionElements$. Here $plays(r, e)$ means that the role r is played by the software system element e.

The essential task of design pattern detection tools is to suggest role assignments. The set of all role assignments identified by a pattern detection tool for a particular schema S and analysed software system P defines a graph with nodes being the entities of P that play roles in S and arcs being the relations between these entities. This graph is referred to as the *projection graph* of S on P, since it "projects" S on the analysed software system P. The undirected projection graph is the graph obtained by disregarding the directions of all edges in the projection graph. That is, it is sufficient to capture which entities interact.

A *candidate* C of a pattern with the schema S is the set of nodes in a connected component of the undirected projection graph of S. That is, a candidate is viewed as an "ensemble" of entities that collaborate to reach the goal represented by the pattern intention.



(A) Relationships



(B) Roles

FIGURE 7.2.1. The schema metamodel

The metamodel of interaction specifications is provided in Figure 7.2.2. The instances of the type formed by the abstract class *SoftwareSystemElement* are software system elements defined in the metamodel from Fig. 7.1.1. Implementations of software system elements at the object level and at the program element level are discussed in Chapter 8.

An Example Schema. As an example, consider the schema that covers the occurrences of collaborational patterns (Section 2.2).

- The role MASTER is played by a non-empty number of objects and represents master objects. A master object owns a non-empty number of field(s) that play the DEPOBJFIELD role and point to the dependent objects, i.e. objects playing the DEPENDENT OBJECT role. A dependent object owns at least one method playing the CALLEE role.
- The role CALLER is played by a non-empty number of methods that are owned by MASTER role players and invoke the methods playing

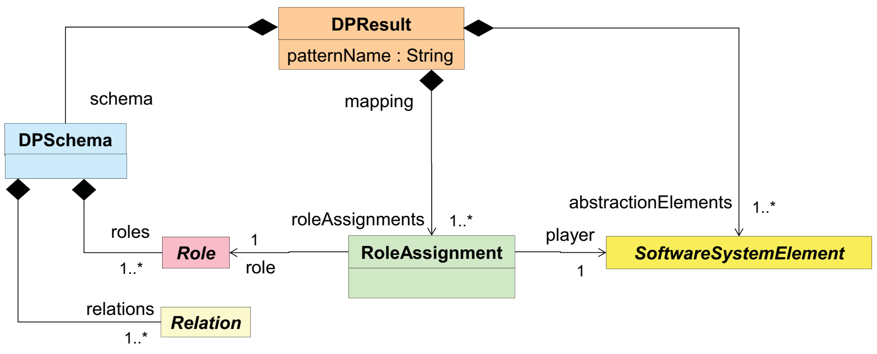


FIGURE 7.2.2. The metamodel of interaction specifications

the CALLEE role players by launching the method calls playing the CALL role.

- The receiver variables of the CALL role players play the RECEIVER role and are dataflow reachable from the corresponding field(s) playing the DEPENDENT FIELD role.

The diagram presented in Figure 7.2.3 describes the collaborational pattern schema. Note that the schemas of other collaborational patterns can be seen as graphs that contain the collaborational pattern schema as a subgraph (with possible changes in role names). For example, in the Observer pattern the role MASTER corresponds to SUBJECT, CALLER corresponds to NOTIFIER, CALLEE corresponds to UPDATE METHOD and so on. Fig. 7.2.4b and Fig. 7.2.4a present the schemas of the Decorator and of the Observer patterns.

An excerpt of the role assignment for an Observer occurrence from JHotDraw 5.1 is presented in Figure 7.2.5.

7.3. The Constraints

This section specifies the core constraints in a FOPL-based language. Each core constraint is represented by a predicate whose parameters are software system elements that play some relevant role(s) in a pattern.

The FOPL language is based on the many-sorted logic (see Meinke and Tuker [49]). The sorts are induced by the kinds of the software system elements and boolean values. In addition to the standard logic symbols $\neg, \vee, \wedge, \Rightarrow, \equiv, \forall, \exists$ the language includes the set of extralogical symbols that specify the set operations $\subseteq, \cup, \cap, \in, \notin$ and equality operations $\neq, =$. A basis for the language is a pair (F, P) where F is the set of function symbols and P is a set of predicate symbols that correspond to the basic functions / relations.

The domain of discourse consists of the software abstraction elements and Boolean values $(true, false)$. The predicate and function symbols are

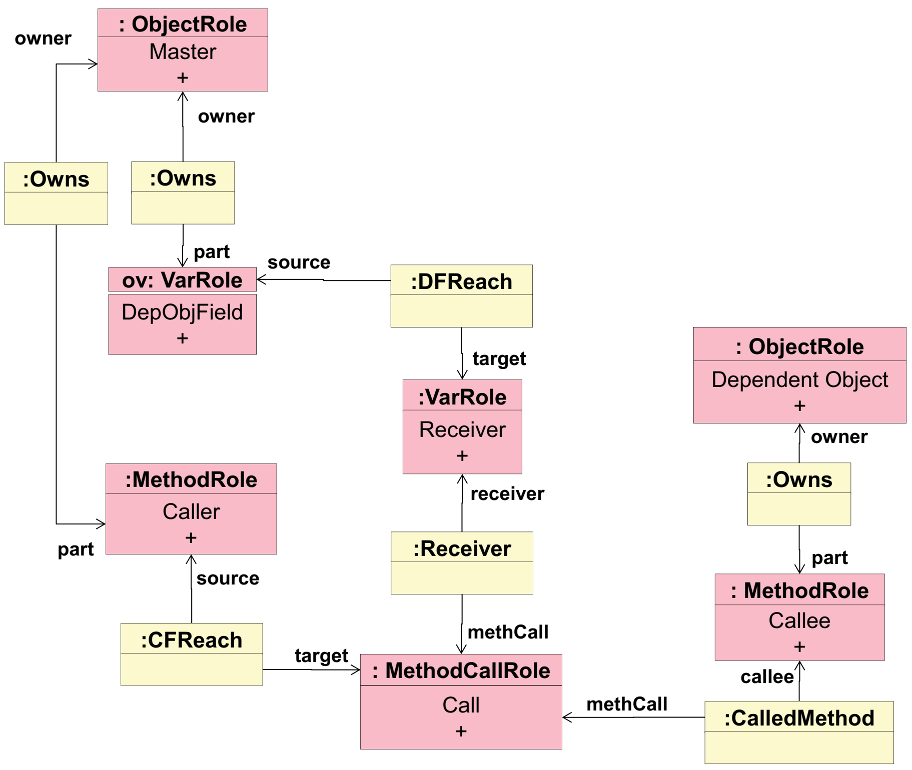


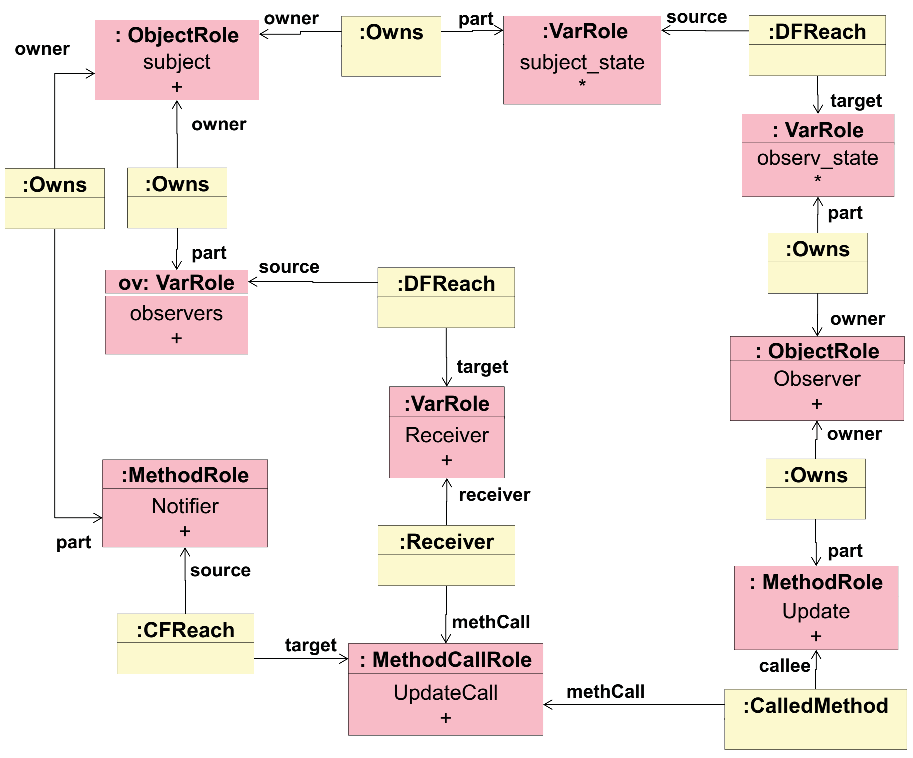
FIGURE 7.2.3. The unified schema of all collaborational patterns. For brevity, each field value is shown at the same position as the respective field name from the schemata metamodel of schemata in Fig. 7.2.1.

mapped to the basic predicates / functions. These are computed from the underlying software system.

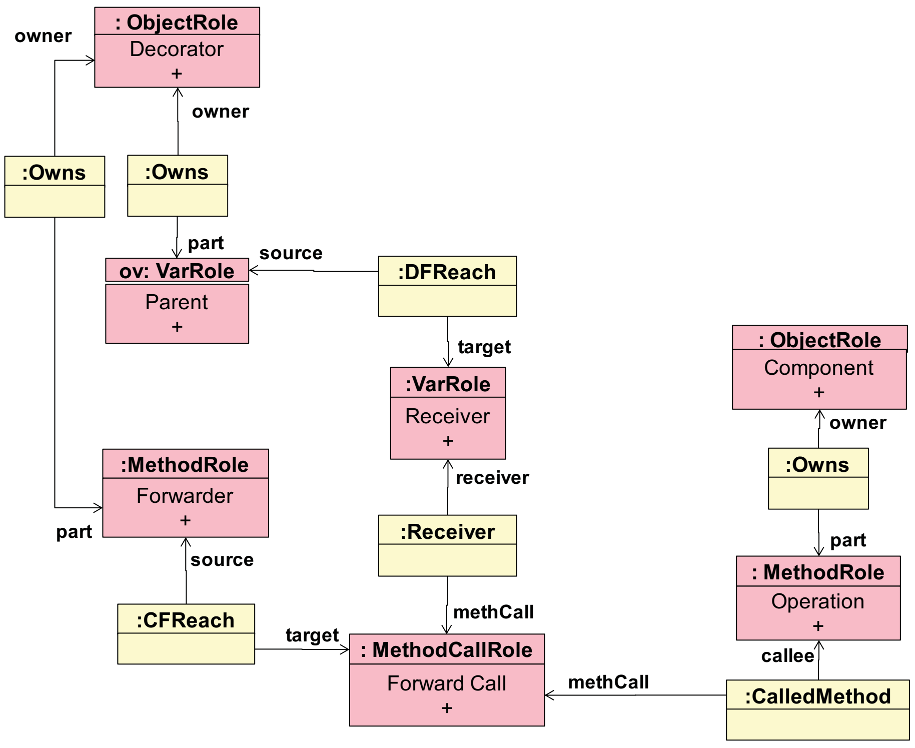
Each constraint is illustrated using a simple code excerpt observed in the benchmarks from Sec. 3.1. In all figures dataflow is depicted by thick pointed lines; control flow is depicted by thick dashed lines. The source code is accompanied with role assignment information. A role is illustrated by an oval box with the role name written within. The thin solid arrow(s) from a box representing a role R lead(s) to the program element(s) playing R. Role players are passed as parameters to the predicates describing the constraints. Each constraint is specified following the informal discussion in Sec. 6.2.

Same Operation. Two different methods M_1, M_2 comprise the input. $SameOperation(M_1, M_2)$ determines whether M_1, M_2 represent conceptually the same operation, following the ideas discussed in Section 6.2.

At the object level the implementation is based on the fact that the methods M_1, M_2 representing the same operation have the same position



(A) The Observer pattern schema is a specialization of the generic collaboration patterns schema. Subject and observer state roles do not have their counterparts in the collaborational pattern schema.



(B) The Decorator pattern schema is a specialization of the generic collaboration patterns schema.

FIGURE 7.2.4. Pattern schemata for collaborational patterns: Two specialisations.

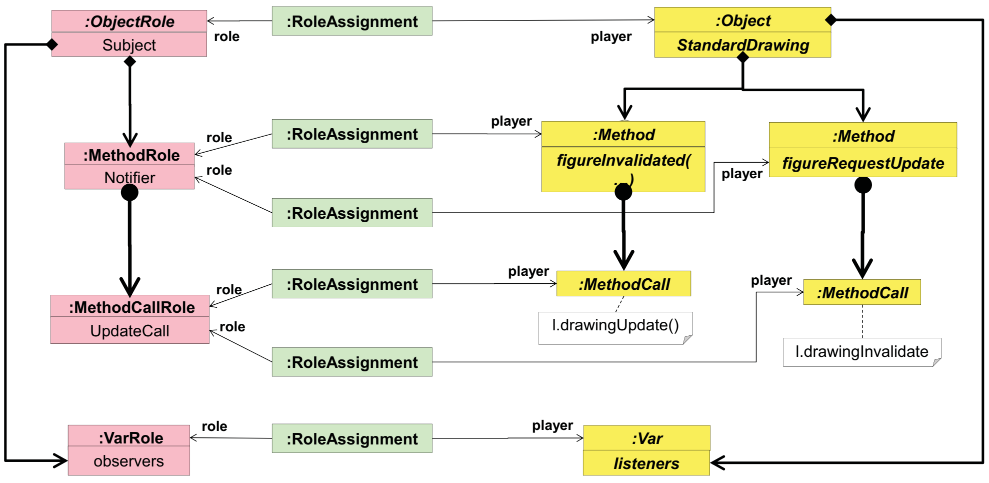


FIGURE 7.2.5. An example role assignment for an Observer occurrence. Instances of the ownership relationship are depicted by thick arrows with black rombs at the source end. Black circles at the source end depict instances of $CFReach$.

in the virtual function tables of the owning classes that have a common superclass.

The static implementation of $SameOperation(M_1, M_2)$ is expressed in terms of method signatures (operation name, parameter types). Two methods M_1 and M_2 possessing a signature s represent the same operation if M_1 belongs to a type T_1 and calls M_2 on an expression of static type T_2 and the two types, T_1 and T_2, have a common supertype T_{CS} that also contains the signature s.

Exclusive Data Flow Source. In the Decorator, Proxy or CoR occurrences, a forwarding method invokes the corresponding method on a *single* "parent" object. Static approximation of forwarding to a single parent can be reduced to ensuring that forwarding is done to objects pointed to by *a single field*. Put another way, if forwarding is done to objects o_1, o_2 pointed to by different fields f_1, f_2 then a given candidate is not a Decorator (resp. CoR, Proxy).

The set of variables $targets$ and the set of variables or objects $sources$ comprise the input. The *Exclusive Data Flow Source* constraint is satisfied if all dataflow from every variable or object V into every variable $t \in targets$ goes through some memory location(s) belonging to $sources$. The intermediate variables are denoted by P.

$$
\begin{aligned}
&ExclusiveDFSource(targets, sources) \equiv \\
&\qquad \exists M \cdot \forall t \in targets \cdot \forall V \cdot \forall P \cdot \exists S \in sources \cdot \\
&\qquad\qquad dfpaths(M, V, t, P) \Rightarrow S \in P
\end{aligned}
$$

To illustrate the constraint, a proper Decorator occurrence (Figure 3.4.2a) and a false Decorator candidate (Figure 3.4.2b) are used. The set of RECEIVER VARIABLES is passed as the parameter to *targets* and the set of PARENT FIELDS is passed as the parameter to *sources*.

Recall that in Decorator, Proxy or CoR occurrences, a forwarding method invokes the corresponding method on a *single* "parent" object. Static approximation of "forwarding to a single parent" is reduced to ensuring that forwarding is done to objects pointed to by *a single field*. This approximation works as follows: if forwarding is done to objects o_1, o_2 pointed to by different fields f_1, f_2 then a given candidate is not a Decorator (CoR, Proxy). That is, the Exclusive Data Flow constraint is satisfied and the set *sources* consists of exactly one field F.[3]

For example, in Figure 3.4.2a $F = myDecoratedFigure$. In the false Decorator candidate from Figure 3.4.2b (which is a Tree Forwarder occurrence, see Section 3.4 for more details) the set *sources* consists of two different fields $myFE1$ and $myFE2$. Forwarding might be done to two different objects pointed to by these fields.

Exclusive Data Flow Target. The set of variables or objects *sources* and the set of variables *targets* comprise the input. The *Exclusive Data Flow Target* constraint is satisfied if all dataflow from every object or variable $s \in sources$ into each variable V goes through some variable(s) $T \in targets$.

$$
\begin{aligned}
&ExclusiveDFTarget(sources, targets) \equiv \\
&\qquad \exists M \cdot \forall s \in sources \cdot \forall V \cdot \forall P \cdot \exists T \in targets \cdot \\
&\qquad\qquad dfpaths(M, V, t, P) \Rightarrow S \in P
\end{aligned}
$$

The intermediate variables are denoted by P.

To illustrate the constraint, a Singleton occurrence (Figure 3.4.3a) and a false Singleton candidate (Figure 3.4.3b) are used. The set of SINGLETON VARIABLES is passed as the parameter to *targets* and the set of SINGLETON OBJECTS is passed as the parameter to *sources*.

In order to limit the maximal number of Singleton objects, the Exclusive Data Flow Target requires all dataflow from newly created Singleton objects to go into a finite number of shared variables (e.g. into the single variable *instance* in Figure 3.4.3a).

[3]In incomplete programs, O might be anonymous.

In the false Singleton candidate from Figure 3.4.3b the constraint fails. The dataflow from newly created objects goes into the non-static variable *newStats* , therefore the maximal number of singleton objects is potentially unlimited.

Field Update. Given a field V owned by the object O, the *Update Field* states that if there exists a method *Setter* owned by O and a variable or an object *source* such that $\mathit{source} \rightarrowtail_{\mathit{Setter}} V$ then *Setter* is not a construction method and can be called (directly of transitively) by a method that does not belong to O:

$$\begin{aligned} UpdateField(V) \equiv \exists \mathit{source} \cdot \exists \mathit{Setter} \cdot \\ \mathit{dfreach}(\mathit{source}, V, \mathit{Setter}) \Rightarrow \\ \neg \mathit{isConstructor}(\mathit{Setter}) \wedge \mathit{ExternallyCalled}(\mathit{Setter}) \end{aligned}$$

To illustrate the constraint, a proper occurrence of the Strategy pattern is contrasted to a false candidate. Note that in every Strategy occurrence the Update Field constraint is satisfied ensuring that the STRATEGY object can be replaced after a CONTEXT object is constructed. In this case, V is a player of the CURRENT STRATEGY role.

In the Strategy occurrence from Figure 3.4.4a the CURRENT STRATEGY field *fUpdateStrategy* is updated in the public non-construction method *setDisplayUpdateStrategy*.

In the false Strategy candidate from Figure 3.4.4b the CURRENT STRATEGY field *myCRIF* is updated only during the construction of an instance of the owning class *RenderableImageOp* so that the Update Field constraint is violated.

State Propagation. The variable sets *Sources* and *Targets* whose variables keep the state of some objects and the method M comprise the input. The *State Propagation* constraint is satisfied if there exists a dataflow (in the scope of M) from each variable belonging to *Sources* into some variable belonging to *Targets* :

$$\begin{aligned} \mathit{StatePropagation}(M, \mathit{Sources}, \mathit{Targets}) \equiv \\ \forall U \in \mathit{Sources} \cdot \exists V \in \mathit{Targets} \cdot \\ \mathit{dfreach}(U, V, M) \end{aligned}$$

The constraint is satisfied in some Observer occurrences where subjects propagate state changes to their observers via dataflow. The set of subject state variables is passed as the parameter to *Sources* , the set of observer state variables is passed as the parameter to *Targets* and a notification method is passed as the parameter to M.

A proper Observer occurrence where *StatePropagation* is satisfied is provided in Figure 3.4.5a. The SUBJECT role is played by objects of the type *StandardDrawing*. Denote a Subject role player by S. The NOTIFIER method is $S.figureInvalidated(...)$. The SUBJECT STATE field is $S.invalidatedRectangle$ represents rectangular areas wrapping changed areas ("invalidated areas").

Denote by o any observer object representing a related drawing view (i.e. an object pointed to by$S.fListeners$). When a drawing is changed, its "invalidated area" is updated; the updates must be propagated to each OBSERVER STATE field $o.fDamage$. This field represents the rectangular area(s) within o that were affected by changes in $S.figureInvalidated$.

The changes in the state of S are propagated to each observer via the dataflow from $S.invalidatedRectangle$ into $o.fDamage$. This dataflow is caused by the actions of the notifier method $S.figureInvalidated(...)$. The changes of a subject S are propagated by means of the dataflow $S.invalidatedRectangle \rightarrowtail e.rectangle$ and $e.rectangle \rightarrowtail o.fDamage$.

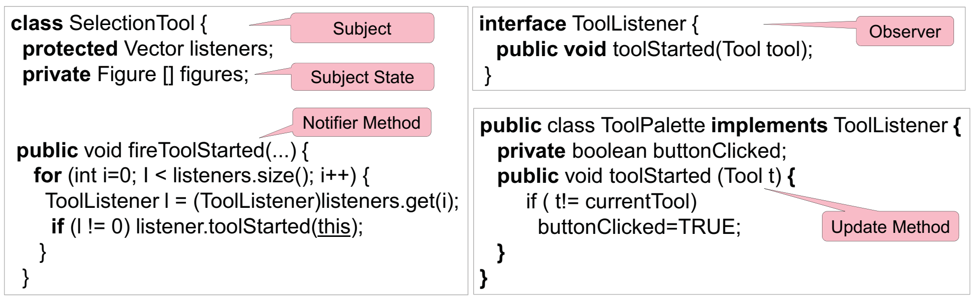
The State Propagation constraint fails in a false Observer candidate in Figure 3.4.5b (which is a *service loop* that iterates over a set of objects, performing some job). Here subjects are instances of the type $ComplexTransform$. Denote a SUBJECT role player by S. The OBSERVERS VARIABLE field $S.transformations$ points to instances of $TransformAST$ (basic transformations) that play the OBSERVER role. Basic transformations related to S are executed sequentially via the NOTIFIER method $S.apply(...)$ that applies the method *update* of each basic transformation. "Subject state" , "observer state" as well as the relevant dataflow do not exist.

Note that propagating state changes via dataflow is too restrictive and recall might be damaged. Therefore the following subsections relax the State Propagation constraint to the Contents Flow constraint and further to the Identity Flow and to the Null Flow constraints.

Contents Flow. First, the necessary terminology is introduced. The object o_S representing the state of the object o might be stored not only in a fields $o.f_{1..k}$ but also in some field owned by objects pointed to $o.f_{1..k}$. In this case o accesses the information about the own state through a sequence of field accesses. In this case it is said that o_S is *transitively referenced* by o.

It is said that *contents flow* from the subject S to the observer O if there are fields $sstate_{1..m}$ in an object transitively referenced by S and fields $ostate_{1..k}$ in an object transitively referenced by O and there is a dataflow from $sstate_{1..m}$ to $ostate_{1..k}$ in the scope of a notification method $S.N$. In this case $sstate_{1..m}$ are accepted as the OBSERVER STATE role players and $ostate_{1..k}$ are accepted as as the SUBJECT STATE role players. The State Propagation constraint is relaxed by letting these roles be played by fields that are not necessarily directly owned in O or S.

Identity Flow and Null Flow. According to [27], the Observer pattern guarantees that "all Observer objects are notified whenever the corresponding subject undergoes state change". However, state changes are not necessarily propagated by means of data or contents flow. It is sufficient that an observer object is *somehow* made aware of the changes in the state



```
class SelectionTool {
  protected Vector listeners;
  private Figure [] figures;

 public void fireToolStarted(...) {
   for (int i=0; I < listeners.size(); i++) {
     ToolListener l = (ToolListener)listeners.get(i);
      if (l != 0) listener.toolStarted(this);
   }
 }
```

```
interface ToolListener {
  public void toolStarted(Tool tool);
 }
```

```
public class ToolPalette implements ToolListener {
  private boolean buttonClicked;
  public void toolStarted (Tool t) {
     if ( t!= currentTool)
       buttonClicked=TRUE;
  }
}
```

FIGURE 7.3.1. Observer with identity flow (Quick UML 2001)

of its subject. Figure 7.3.1 illustrates such a scenario from Quick UML 2001. Subjects are instances of $SelectionTool$ that notify associated instances of $ToolPalette$ (i.e. observers). When a tool is started, the method $fireToolStarted$ notifies the observers by invoking the method $toolStarted(Tool)$ on each associated $ToolPalette$ instance, passing itself as a parameter. A palette thus becomes aware of the start event and of its sender and updates appropriately its "clicked" flag. This process is referred to as the *Identity Flow*.

Moreover, some proper Observer occurrences were observed in which a subject notifies its observers without passing any parameters to the update method (i.e. *the Null Flow*).

Note that the Identity Flow and the Null Flow are still optional constraints; that is, there are correct Observer occurrences where these constraints are not fulfilled. In the future, using of additional sources of information (e.g. program element names) is suggested in order to increase the confidence scores of such candidates. More details are provided in Section 4.6.

Ordering of Control - and Data Flow. For the sake of simplicity, some auxiliary concepts are first introduced.

Setter Call. The method M forming the scope of the dataflow into the variable V is referred to as *V-setter*. The relation $setter\_call \subseteq MethodCalls \times Variables$ is defined. It is written $setter\_call(mc, v)$, meaning that the method call mc refers to the v-setter. This relation is defined as follows:

$$
\begin{aligned}
&setter\_call(mc, v) \equiv \\
&\qquad\qquad \exists M \cdot \exists u \cdot calledMethod(mc, M) \wedge dfreach(u, v, M)
\end{aligned}
$$

The method M, the variable V and the set of method invocations issued by M comprise the input. The Interleaving Order constraint[4] is satisfied if $CFG_{abstr}(M)$ satisfies the following conditions:

(1) For each invocation vc of a V-setter, at least one method call $mc \in Invocations$ follows vc.
(2) There exists some V-setter call vc_{start} such that no method invocation $mc \in Invocations$ precedes vc.

$$\begin{aligned}&InterleaveOrder(M, V, Invocations) \equiv \\ &\quad \forall vc \cdot \exists mc \in Invocations \cdot setter\_call(vc, V) \Rightarrow cfreach(M, vc, mc) \\ &\qquad\qquad\qquad\qquad \wedge \\ &\qquad \exists vc_{start} \in Invocations \cdot \forall mc \in Invocations \cdot \\ &\qquad\qquad setter\_call(vc, V) \wedge \neg cfreach(M, vc, mc_{start})\end{aligned}$$

To illustrate the constraint, a State occurrence and a false State candidate are used. An Action method is passed as the parameter to M , a Current State field is passed as the parameter to V and the set of Action Call invocations invoked by M is passed as the parameters to $Invocations$. The Interleaving Order constraint is satisfied in State occurrences and means that 1) every update of the current state is followed by several action method invocations and 2) the "state machine" is started by setting the initial state.

In a State occurrence from Fig. 3.4.6a the setter method is *setTool*. The Launcher method *mousePressed(...)* sets the current state by calling *setTool(getSelectedTool(...)* and then invokes the next Action method by calling *(getTool.mouseDown()*. Several Action methods (*mouseDown*, *mouseDrag* and *mouseUp*) are called until the current state is again updated by setting back the default tool *setTool(getDefaultTool())*.

In the false State candidate (which is a Strategy occurrence) from Figure 3.4.6b, the Interleaving Order constraint fails since the Action method *itemChanged(Event e)* and any current state updates within *addListener()* might be called in any order.

Uniquely Created Product Type. The set of methods *CreatorMethods* yield product objects that belong to the types *ProductTypes*. The sets *CreatorMethods* and *ProductTypes* comprise the input. Denote by CM_1, CM_2 any two different creator methods. Denote by $ProductVar_1$, $ProductVar_2$ the variables into which product objects flow provided the dataflow is in the scope of CM_1 (respectively, of CM_2). The points-to sets of $ProductVar_1$

[4]I do not address Sequential Order in this thesis since no examples of its applicability were found in the examined benchmarks.

and $ProductVar_2$ are said to be *type-disjoint* if

$$\begin{aligned}&\forall P_1 \in pointsTo(CM_1, ProductVar_1)\cdot\\ &\forall P_2 \in pointsTo(CM_2, ProductVar_2)\cdot\\ &\qquad\qquad type(P_1) \neq type(P_2)\end{aligned}$$

The *Uniquely Created Product Type* (UCPT) constraint is satisfied if the points-to sets of $ProductVar_1$ and $ProductVar_2$ are said to be type-disjoint.

First several auxiliary predicates are introduced. The predicate $inducedDF \subseteq Types \times Vars \times Methods$ describes the flow of objects of a given type(s) in the scope of a given method.

It is written $inducedDF(ProductTypes, ProductVar, Method)$ if all objects of the types $ProductTypes$ flow into $ProductVar$ and the dataflow is formed by $Method$. Formally,

$$\begin{aligned}&inducedDF(ProductTypes, ProductVar, Method) \equiv\\ &\qquad \forall Prod : type(Prod) \in ProductTypes\cdot\\ &\qquad\qquad dfreach(Prod, ProductVar, Method)\end{aligned}$$

The predicate $type\_disjointPT \subseteq Vars \times Vars$ describes type-disjoint points-to sets.

$$\begin{aligned}&type\_disjoint(Var_1, Var_2) \equiv\\ &\quad \forall P_1 \in pointsTo(ProductVar_1) \cdot \forall P_2 \in pointsTo(ProductVar_2)\cdot\\ &\qquad\qquad type(P_1) \neq type(P_2)\end{aligned}$$

The UCPT constraint is specified as follows:

$$\begin{aligned}&UCPT(CreatorMethods, ProductTypes) \equiv\\ &\qquad \forall CM_1, CM_2 \in CreatorMethods\cdot\\ &\qquad \forall ProductVar_1 \cdot \forall ProductVar_2\cdot\\ &\qquad inducedDF(ProductTypes, ProductVar_1, CM_1)\wedge\\ &\qquad inducedDF(ProductTypes, ProductVar_2, CM_2)\\ &\qquad\qquad \Rightarrow\\ &\qquad\qquad type\_disjoint(ProductVar_1, ProductVar_2)\end{aligned}$$

To illustrate the constraint, an Abstract Factory occurrence and a false Factory Method candidate are used. The set of FACTORY METHODSs is passed as the parameter to $CreatorMethods$, the set of CONCRETE PRODUCT types is passed as the parameter to $ProductTypes$.

In the Abstract Factory occurrence from Fig. 3.4.11a the constraint is satisfied since different creator methods (i.e. the creator methods with the same signature that belong to different concrete factories ($BufferParserFactory.getInputStream()$, $FileParserFactory.getInputStream()$) create products of different types ($BufferedInputStream$ and $FileInputStream$ respectively).

In the false Factory Method candidate from Fig. 3.4.11b the constraint is failed since different creator methods (i.e. the creator methods with the same signature that belong to different concrete factories $ComponentFigure.handles()$ and $RectangleFigure.handles()$) create products of the same type, namely $HandleEnumerator$.

The *single-creator scenarios* provided in Stencel et al. [64] and in Gamma et al. [27, Page 125] represent very rare scenarios where the single creator method *Creator* creates products of different types by passing different parameters to the same (single) creator method in different executions. The UCPT constraint provided in this section can be seen as a particular case of the concept discussed in Section 6.2.1 where any two invocations to the same creator method that carry different parameters must yield product objects of different types. In order to deal with the single-creator scenarios, the context-sensitive dataflow analysis (see Nielson et al. [51]) must be used. However, the single-creator scenarios were not found in the examined repositories and therefore are not supported in the UCPT specification in this section. The dataflow analysis supported in the implementation of the Pattern Interaction Framework is context-insensitive. See Section 8.1 for more details.

Sibling Creation. The field V and the caller method $Caller$ that invokes callee method(s) on the receiver objects pointed to by V comprise the input. Denote by O the object owning $Caller$. Let the function $owner(P)$ return the owner object of a field/method P.

The Sibling Creation constraint is satisfied if each non-anonymous object pointed to by V belongs to a type that are siblings to $type(O)$. Formally,

$$\begin{aligned} &SiblingCreation(Caller, V) \equiv \\ &\qquad \forall o_V \in pointsTo(V) \setminus \{anonymous\} \cdot \exists T \cdot \\ &\qquad subtype(type(o_V), T) \wedge subtype(type(Owner(Caller), T)) \end{aligned}$$

To illustrate the Sibling Creation constraint, an occurrence of Dynamically Typed Proxy and an occurrence of Decorator are used. A FORWARDER method is passed as the parameter to *Caller* and a PARENT field is passed as the parameter to V. The Sibling Creation constraint is satisfied in Dynamically Typed Proxy occurrences, indicating that forwarding can be done only to the objects whose type is a sibling of the proxy type.

In the occurrence of Dynamically Typed Proxy from Figure 3.4.7a, the Sibling Creation constraint is satisfied since the types of all objects pointed to by the Real Subject object (*HandleTracker*, *DragTracker*, *AreaTracker*) are siblings of *SelectionTool*.

In the occurrence of Decorator from Figure 3.4.7b, the Sibling Creation constraint fails since the points-to set of the Parent field includes only the anonymous object.

Partial Coverage / Full Coverage. The variable V and the caller method *Caller* that invokes the method(s) *Callees* on the objects pointed to by V comprise the input. The Full Coverage constraint is satisfied if for each set P abstracting some path through the control flow of *Caller*[5] and for each object o pointed to by V there is an invocation mc of a callee method whose receiver variable points to o.

Formally,

$$\begin{aligned}&\mathit{FullCoverage}(\mathit{Caller}, \mathit{Callees}, V) \equiv\\ &\qquad\forall P : \mathit{cfpath}(\mathit{Caller}, P) \cdot \forall o \in \mathit{pointsTo}(V) \cdot\\ &\qquad\exists mc \in P \cap \{mi \mid \exists cl \in \mathit{Callees} \cdot \mathit{calledMethod}(mi, cl)\} \cdot\\ &\qquad\qquad o \in \mathit{pointsTo}(\mathit{receiver}(mc))\end{aligned}$$

.

The Partial Coverage constraint is satisfied if:

- There is a set P_t abstracting some path through the control flow of *Caller* that contains an invocation mc of some $\mathit{Callee} \in \mathit{Callees}$ on each object $o \in \mathit{pointsTo}(V)$.
- There is a set P_p abstracting a path corresponding to a diverting execution, i.e. some path through the control flow of *Caller* that contains no invocation of any $\mathit{Callee} \in \mathit{Callees}$ on some object $o \in \mathit{pointsTo}(V)$.

Formally,

$$\begin{aligned}&\mathit{PartialCoverage}(\mathit{Caller}, \mathit{Callees}, V) \equiv\\ &\qquad\exists P_t : \mathit{cfpath}(\mathit{Caller}, P_t) \cdot \forall o \in \mathit{pointsTo}(V) \cdot\\ &\qquad\exists mc \in P_t \cap \{mi \mid \exists cl \in \mathit{Callees} \cdot \mathit{calledMethod}(mi, cl)\} \cdot\\ &\qquad\qquad o \in \mathit{pointsTo}(\mathit{receiver}(mc))\\ &\qquad\qquad\wedge\\ &\qquad\exists P_p : \mathit{cfpath}(\mathit{Caller}, P_p) \cdot \exists o \in \mathit{pointsTo}(V) \cdot\\ &\qquad\forall mc \in P_p \cap \{mi \mid \exists cl \in \mathit{Callees} \cdot \mathit{calledMethod}(mi, cl)\} \cdot\\ &\qquad\qquad o \notin \mathit{pointsTo}(\mathit{receiver}(mc))\end{aligned}$$

To illustrate the Full Coverage constraint, a Decorator occurrence from Fig. 3.4.7b and an Observer occurrence from Fig. 3.4.5a are used. A caller

[5]See the corresponding definition in Section 7.1.

method (a FORWARDER method, respectively a NOTIFIER method) is passed as the parameter to *Caller*. The set of callee method(s) invoked by *Caller* (the OPERATION methods; respectively the UPDATE methods) is passed as the parameters to *Callees* and a variable(s) pointing to the dependent objects (the PARENT fields, respectively the OBSERVERS fields) is passed as the parameter to V. In both cases, in each execution of *Caller* some callee method $Callee \in Callees$ is invoked on each dependent object.

To illustrate the Partial Coverage constraint, a CoR occurrence from Fig. 3.4.8 and a false Observer candidate from Fig. 3.4.9 are used. In both cases, the Partial Coverage constraint is satisfied. A caller method (a FORWARDER method, respectively a NOTIFIER method) is passed as the parameter to *Caller*; the set of callee method(s) invoked by *Caller* (the OPERATION methods, respectively the UPDATE methods) is passed as the parameters to *Callees* and a field(s) pointing to the dependent objects (the PARENT field, respectively the OBSERVERS field) is passed as the parameter to V. In both cases, *Caller* does not invoke some callee method on each dependent object in some its execution. In the CoR pattern this happens since forwarding is not done at all in some execution. In a false Observer candidate this happens because a callee method is not invoked on some dependent object in some loop iteration.

In the static implementation of the Pattern Specification Framework provided in this thesis (Section 8.3) the Partial and Full Coverage constraints are implemented by enumerating individual paths in the control flow of *Caller*. This approach does not require computing of points-to sets and is therefore more efficient.

7.4. Behavioral Pattern Specifications

This section specifies several collaborational patterns (Decorator, Proxy, CoR, Tree Forwarder, Observer, Composite) formally by restricting the role players of the collaborational schema (Fig. 7.2.3). This is done by applying the appropriate constraints from Section 7.3.

The subsections of this section provide several auxiliary predicates, several specification examples and show the applicability of this approach for the remaining patterns.

Auxiliary predicates. The auxiliary predicates represent queries on the role assignments built for the collaborational schema.

The caller methods associated with a given master object. The function $callersOf(CallerRole, m)$ returns the set of caller methods owned by a master object m and playing the role *CallerRole*:

$$callersOf(CallerRole, m) = \{caller \mid plays(CallerRole, caller) \wedge \\ method(m, caller) \\ \}$$

Consider the Observer pattern as an example. For a subject s the invocation
$callersOf(Notifier,\ s)$ returns the set of notifier objects fields owned by s.

Dependent objects and the fields pointing to dependent objects. The function $depObjFieldsOf(FieldRole,\ m)$ returns the set of fields owned by a master object m and playing the role $FieldRole$:

$$depFields(FieldRole, m) = \{pf \mid field(m, pf) \wedge plays(FieldRole, pf)\}$$

Consider the Observer pattern as an example. For a subject s the invocation
$depObjFields(observers,\ s)$ returns the set of observer fields owned by s.

The function $depObjectsOf(FieldRole,\ m)$ returns the set of objects dependent on a master object m , i.e. objects pointed to by the fields playing the role $FieldRole$ that are owned by m :

$$depObjects(FieldRole, m) = \bigcup_{df \in depFields(FieldRoleName, m)} pointsTo(df)$$

The receiver variables of the method calls issued by a given method. The function $recVarsOf(CallRole, caller)$ returns the set of variables that play the RECEIVER role and serve as the receiver variable for any method call that plays the role $CallRole$ and issued by a given caller method $caller$.

$$\begin{aligned} recVarsOf(CallRole, caller) = \{rv \mid \exists c :& cfreach(caller, c) \wedge \\ & plays(CallRole, c) \wedge \\ & rv = receiver(c) \wedge \\ & plays(Receiver, rv) \\ \}& \end{aligned}$$

Consider the Composite pattern as an example. For a composite object $comp$ the invocation $recVarsOf(ForwardingCall, forwarder)$ returns the set of receiver variables of all forwarding calls issued by a given forwarder method $forwarder$.

Callee methods . The function $calleesOf(CalleeRole, caller)$
returns the set of methods playing the role $CalleeRole$ that are invoked by a method $caller$:

$$\begin{aligned} calleesOf(CalleeRole, caller) \equiv \{ & m \mid \exists call \cdot cfreach(caller, call) \wedge \\ & calledMethod(call, m) \wedge \\ & plays(CalleeRole, m) \\ \} & \end{aligned}$$

Consider the Observer pattern as an example. For a notifier method n the invocation
$calleesOf(UpdateMethod,\ n)$ returns the set of update methods invoked by n.

Specifications of several patterns. The core constraints specified in Section 7.3 were defined on software abstraction elements. This section specifies them at role level by linking the roles to the elements that play them.

The specifications for the Observer, Composite, CoR, Decorator and Proxy patterns are provided. Each subsection within this section specifies only one pattern in the corresponding group of structurally similar patterns. Other patterns can be specified by renaming role names in the corresponding schemas and by following the explanations in the sections.

Throughout this section, the following denotations are used:

- m is a master object,
- c is a caller method owned by m
- df is a field pointing to the dependent objects
- u is a callee method invoked by c
- do is a dependent object dependent from m
- ms (resp. dos) are the variable(s) storing the state of m (resp. of do)

Observer, Composite. In the specification of the Observer pattern, the core constraints restrict the players of the roles in the Observer schema (Fig. 7.2.4a). The constraints are applied as follows:

- A notifier method c and each updated method invoked by c must represent different operations
- Each field pointing to observers must be updated after its owner is constructed
- A notification method must call an update method on each dependent observer in each own execution
- Subject state changes are propagated to all dependent observers.

Formally,

$$\begin{aligned}
&\forall m\, \forall c \cdot plays(Subject, m) \wedge c \in callersOf(Notifier, m) \\
&\wedge \\
&\forall u \cdot u \in calleesOf(UpdateMethod, c) \wedge \neg SameOperation(c, u)) \\
&\wedge \\
&\forall df \cdot plays(Observers, df) \wedge UpdateField(df) \\
&\wedge \\
&FullCoverage(c, calleesOf(UpdateMethod, c), depFields(Observers, m) \\
&\wedge \\
&\forall do \cdot do \in depObjectsOf(Observers, m) \wedge \\
&StatePropagation(c, \\
&\{ms \mid plays(SubjectState, ms) \wedge transRef(m, ms, c)\}, \\
&\{dos \mid plays(ObserverState, dos) \wedge transRef(do, dos, c)\})
\end{aligned}$$

The application of the core constraints in the specification of the Composite pattern differs in the following points:

(1) The application of *SameOperation* is positive since the action method owned by a container object and the operation method owned by the corresponding part object represent the same operation.
(2) The constraint *StatePropagation* is not applied.

Decorator, CoR, Proxy. In the specification of the Decorator pattern, the core constraints restrict the players of the roles in the Decorator schema (Fig. 7.2.4b). The constraints are applied as follows:

- A forwarder method c and each operation method invoked by c must represent the same operation
- Each parent field must not be updated after its owner is constructed
- A forwarder method must call an operation method on the parent object in each execution
- Forwarding can be done to any object and not only to those whose type is a sibling of the decorator type.

Formally,

$$\begin{aligned}
&\forall m\,\forall c \cdot plays(Decorator, m) \wedge c \in callersOf(Forwarder, m)\\
&\qquad\wedge\\
&\qquad\forall u \cdot u \in calleesOf(Operation, c) \wedge SameOperation(c, u))\\
&\qquad\wedge\\
&\qquad\forall df \cdot plays(ParentField, df) \wedge\\
&\qquad\quad \neg SiblingCreation(c, df) \wedge \neg UpdateField(df)\\
&\qquad\wedge\\
&\qquad FullCoverage(c, calleesOf(Operation, c),\\
&\qquad\quad depFields(ParentField, m)
\end{aligned}$$

The CoR specification differs from the Decorator specification only in one point: the constraint $PartialCoverage$ is enforced instead of $FullCoverage$.

The Proxy specification differs from the Decorator specification in two points:

(1) Apply $SiblingCreation$ positively instead of negating it.
(2) Drop $FullCoverage$.

Specifying the remaining patterns - An overview. For the sake of simplicity, the previous section does not specify all patterns mentioned in [27]. However, all collaborational (vs. creational) patterns can be specified also by using the constraints from Section 7.3 and restricting the players of the roles in the collaborational (vs.creational) schema. For example, Figure 7.4.1 summarizes the interaction specifications for the collaborational patterns. Individual patterns (in the rows) are associated to the constraints used in specifications of these patterns (in the columns). The cells may contain one of the following symbols:

- $\surd$ (resp. X) - positive vs. negative application, like Same or Different Operations
- *total* or *partial* - for Total vs. Partial Coverage.

The activity diagram in Figure 7.4.2 shows the order of constraint application when specifying the colalboration patterns. An activity corresponds to a pattern detection module. A branch corresponds to a constraint application.

When the boxes standing for certain activities are wrapped by thick double line, this means that additional information (e.g. program element names) is needed to further distinguish between the patterns mentioned in the box corresponding to this activity. For example, Adapter and Command can be distinguished by analyzing comments to reveal additional clues regarding the intention. 1:1 Observer can be distinguished from Bridge and Strategy by analyzing program element names. For example, the names

	Same Op	Unique DF Source	Field Update	Coverage	Sibling Creation	StatePropagation	CF-DF Order	Un.Cr.Prod.Type	Excl.DF Target
Abstract Factory	X							✓	
Bridge	X	✓	✓						
Composite	✓		✓	total					
CoR	✓	✓	X	partial					
Decorator	✓	✓	X	total					
Factory Method	X							✓	
Mediator	X		✓						
Memento	X						✓		
Observer	X		✓	total		✓			
Proxy	✓	✓	X		✓				
Prototype	X							✓	
Singleton	X								✓
State	X	✓	✓				✓		
Strategy	X	✓	✓						
Tree Forwarder	✓	X		total					

FIGURE 7.4.1. Constraints and patterns

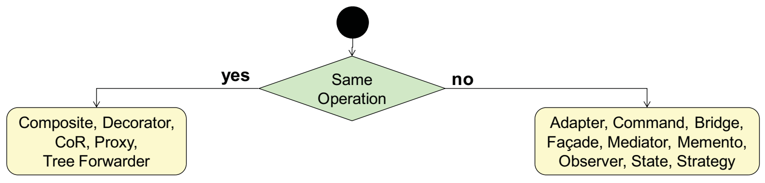
of Observer classes usually contain the word “listener” and “observer”. The names of notifier methods usually contain the words “fire” or “notify”.[6] Bridge can be distinguished from Strategy by observing whether new subclasses were massively added/deleted in Master classes.

7.5. Summary

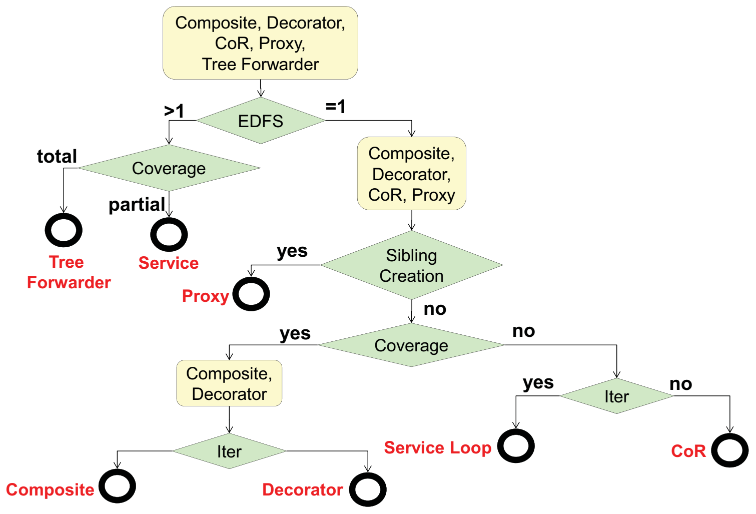
The basic relations and functions presented in Section 7.1 shield solution descriptions from the program analyses employed by a particular pattern detection tool. Figure 7.5.1 illustrates “shielding” visually.

The thick solid vertical arrow illustrates the abstraction level. Interaction specifications occupy the top level. Roles, relationships and the core constraints occupy the next level. The relations/functions serve as the building bricks for the constraints and obtained the next level. The relations and functions separate the generic-level software system descriptions from the program analysis specific descriptions: implementation variants (at the program element level) and program trace descriptions (at the object level). The separation of the generic level from the program-analysis-specific level is illustrated by the thick dashed line.

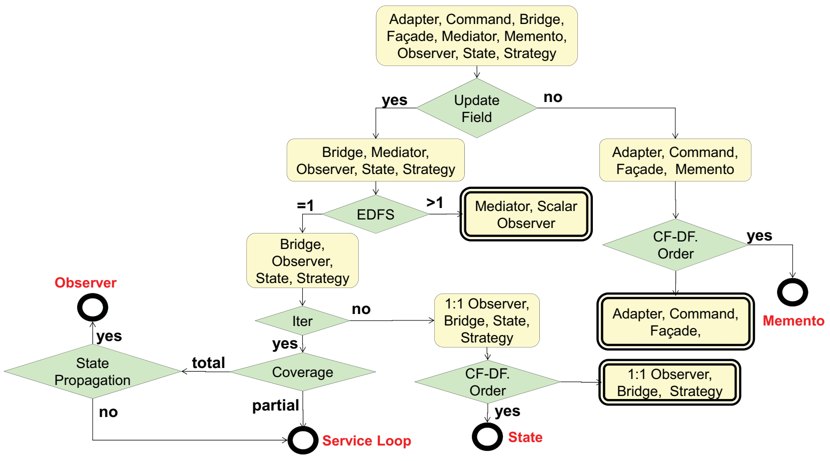
[6]In essence, name conventions used in 1:1 Observers are stronger than in usual Observers because designers want to emphasize their intent, distinguishing 1:1 Observers from a lot of structurally similar constructs (Adapter, Command, Strategy etc).


Same Operation
yes
no
Composite, Decorator, CoR, Proxy, Tree Forwarder
Adapter, Command, Bridge, Façade, Mediator, Memento, Observer, State, Strategy

(A) Discrimination of collaborational patterns by the "Forwarding" constraint



(B) Caller and Callee are same operations


Adapter, Command, Bridge, Façade, Mediator, Memento, Observer, State, Strategy
Update Field
yes
no
Bridge, Mediator, Observer, State, Strategy
Adapter, Command, Façade, Memento
EDFS
=1
>1
Mediator, Scalar Observer
Bridge, Observer, State, Strategy
CF-DF. Order
yes
Memento
Adapter, Command, Façade,
Iter
no
1:1 Observer, Bridge, State, Strategy
yes
Coverage
total
partial
State Propagation
yes
Observer
no
Service Loop
CF-DF. Order
yes
State
1:1 Observer, Bridge, Strategy

(C) Caller and Callee are different patterns

FIGURE 7.4.2. Detection of collaborational patterns

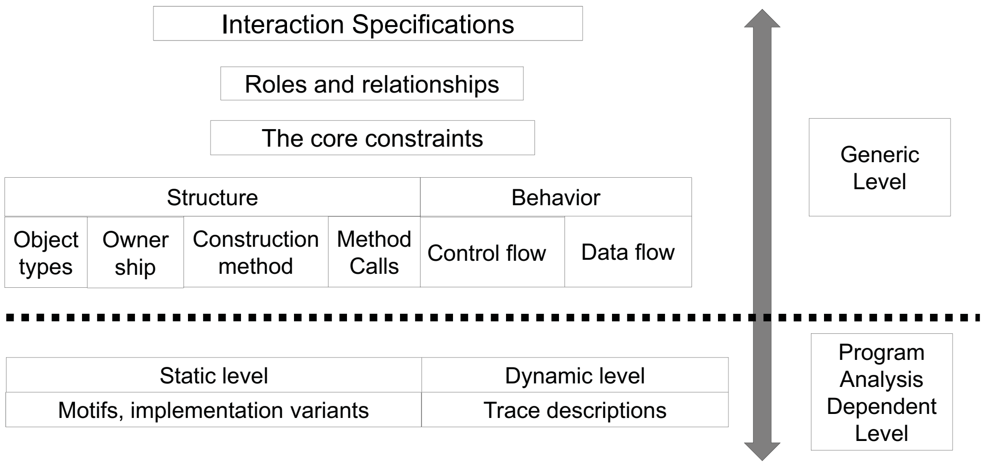


FIGURE 7.5.1. Formal DP specifications - Summary

Eventually, program-analysis-specific descriptions capture a number of corresponding software artifacts: implementations (at the static level) or program traces (at the dynamic level).

Thus the Pattern Interaction Framework provides a common basis to allow researchers from various DP communities to meet, to discuss (and to resolve) conceptual disagreements and then focus on the best way to mix and match detection techniques for the specified concepts. Chapter 4 describes the experiments on mixing of detection techniques. The outputs of pattern detection tools implementing different techniques were fused, obtaining improvements in accuracy.

CHAPTER 8

Implementation Issues

It was decided to implement the Pattern Interactions Framework (Chapter 7) using a static program approximation because a dynamic approximation does not cover all relevant behavior (Section 2.1) and might be extremely inefficient in the case of big benchmarks. However, full-scaled static behavioral analyses tend to be inefficient, especially in object-oriented languages with polymorphism and dynamic method binding (Nielson et al. [51]). Therefore the Pattern Interactions Framework is implemented using **the limited version of the behavioral analysis** that covers only necessary pattern-relevant features and delivers very high precision and high recall. Rare and exotic pattern implementation variants might be still missed. Conceptually, *the powers of several pattern detection techniques* are combined in a single pattern detection tool.

The data fusion approach presented in Chapter 4 combines the powers of several pattern detection techniques - *by fusing the outputs* of the pattern detection tools employing these techniques. However, it was decided to depart from the data fusion approach. The following observations (confirmed by the talks with the authors of the tools reviewed in Chapter 3) led to this decision:

- Making the authors reimplement their tools so as to comply with a common exchange format seems to be unfeasible in this moment and is beyond the time frame allocated for my Ph.D. track.
- The accuracy of the combined result might suffer from the bugs in the involved tools.
- The overall response time is at least the response time of the slowest tool (if the tools are executed in parallel). The situation is aggravated when some pattern tools employ unnecessary or too inefficient program analyses (like the transitive closure of the data flow graph built in [64]).

Section 8.1 explains how to implement the Pattern Interaction Framework using dynamic- and static program analyses. The main principles of the program-level implementation are provided below:

(1) To achieve high precision, the core constraints are used.
(2) To achive high recall, overly restrictive constraints imposed by other tools for precision are relaxed. These relaxations are discussed in Section 8.2.

(3) To achieve high response time, fast structural analyses (e.g. ownership) are executed first. In addition, behavioral analysis scopes are limited. See Section 8.3 for more details.

Section 8.4 discusses several aspects of combining static- and dynamic analyses when implementing the core constraints.

Section 8.5 presents the architecture of a static pattern detection tool DPJF that represents the implementation of the Pattern Interaction Framework.

It should be noted that the current version of DPJF supports only 5 patterns (Decorator, CoR, Proxy, Observer and Composite). So certain constraints (Exclusive Data Flow Target, Control/Data Flow Ordering and Uniquely Created Product Type) that are not used in the specifications of these patterns are not implemented. See Section 8.5 for further details and Section 7.4 for the specifications of the implemented patterns.

8.1. Implementing the Pattern Interaction Framework

This section demonstrates how to map the software abstraction and the generic relations of the Pattern Interaction Framework from Chapter 7 onto the object level and onto the program element level.

Mapping the domain of discourse. At the object level, software abstraction elements are mapped into the *program trace elements* whereas at the static level they are mapped into the *program abstraction elements* that describe their object-level counterparts.

Types, variables and methods are mapped straightforwardly. At the object level types are modeled by RTTI structures that represent groups of objects; at the program element level types are represented by type descriptions.[1]

At the object level methods are modeled by structures keeping the information about relevant events and properties. At the program element level, methods are modeled by their descriptions written in a program language.

At the object level, variables are modeled by memory locations. At the program element level, variables are represented by corresponding variable declarations. In addition, at the program element level a method M that returns a result is associated with the pseudo-variable $return_M$ that keeps this result. In addition, $return_M$ serves as a node in the data flow graph of M (see also Section 8.1).

At the object level, an object is a memory location belonging to some group (i.e. type). At the program element level, an object is represented by the corresponding object abstraction: a creation statement (e.g. $new(...)$ or an anonymous object.

[1]Only class-based languages are used so a type is formed by a class and its superclasses

Abstract element	Object level	Program element level
Object	Object	Object abstraction (anonymous object, new statement)
Variable	Memory slot	Variable description (field, local variable, parameter description)
Type	Runtime type information	Type description by subclassing
Method	Method structure	Method description
Method call	Method call event	Method call statement
Method start/exit points	Start / exit events	Start/exit pseudonode in CFG
Null check	Null check event	Null check expression
		Loop entry statement

FIGURE 8.1.1. Mapping of the domain of discourse

At the object level, method calls are modeled by corresponding events. At the program element level, method calls are modeled by method call statements. Several calls might be represented by one statement (for example, in a loop).

At the object level the start/exit points of a method M correspond to Start/Finish Method events. At the program element level these are represented by the corresponding pseudo-nodes in the control flow graph of M.

Finally, at the program element level yet one pattern-relevant statement is introduced: loop entry statements. These are needed to distinguish only those paths which actually enter loops and thus can contain conceptually relevant method calls.

Figure 8.1.1 summarizes the mapping of the domain of discourse into the object level and into the program element level.

Implementing Generic Relations. Section 8.1 describes the implementations of the generic structural relations at the object level and at the program element level. The remaining sections describe the implementations of the generic behavioral relations: control flow (Section 8.1) and data flow (Section 8.1).

Structural relations. Variable / method ownership and "constructor method" function are modeled straightforwardly by the retrieval operations on the underlying abstraction. Note also a subtlety in the implementations of the relation $calledMethod(Call, Method)$. At the object level, a given method call refers to *a single method*. At the program element level, *a non-empty set of called methods* is computed according to the type(s) of object(s) pointed to by $receiver(mc)$. Tip et al. [66] reviews the corresponding techniques.

At the program element level the transitive subtyping relation is supported. In practice, only a limited number of intermediate classes in the subtyping relation is supported; see Section 8.3 for more details.

Control Flow. At the object level, the abstracted control flow relation is a total order of pattern-relevant events (method calls, null checks, method start /exit points). It is built by recording all trace events and selecting only pattern-relevant ones.

At the program element level, the control flow relation of M is a graph. Its nodes are the static level counterparts of the members of $ExecutionItems$ (Sec. 7.1). An edge between two nodes s, t occurs if t follows s in the control flow graph of M (even if certain pattern-irrelevant events occur between s and t). The net effect of dataflow-related events (e.g. assignments) that are not present in the abstracted control flow relation is compensated in the dataflow relation.

The control flow graph of M consists of:

- Nodes: statements / expressions in the body of M.
- Edges. *Intraprocedural* edges cover control structures in the body of M and the edges from the return statement(s) of M to the exit point of M. For each method invocation mi in the body of M, there is an *interprocedural* edge from mi to $Start_{calledMethod(mi)}$ and from $Exit_{calledMethod(mi)}$ to mi.
- The CFGs of method(s) invoked from M.

Data Flow.

Dataflow graph. At the object level, the dataflow relation is a total order and is computed by tracking the *dataflow-related events* in the control flow, i.e. assignments of objects to variables, method parameter passing and returning of results.

At the program element level, the dataflow relation of a method M is a partial order. Its nodes are variables and objects. Edges are formed by assignments statements, method parameter passing and method returning statements.

The static dataflow analysis is performed in the subset-based way (i.e. a statement $x := y$ yields the constraint $pointsTo(y) \subseteq pointsTo(x)$) and is categorized as follows:

- *Flow-insensitive* - the order of individual statements is not relevant.
- *Context-insensitive* - individual calling contexts of the same method are not distniguished.
- *Field-sensitive* - only in the Dynamically Typed Proxy (Section 3.4) context. Different objects in the points-to set of the parent field (and the invocations of callee methods addressed to these objects) are distinguished. This is done in order to distinguish Dynamically Typed Proxy implementations from structurally similar CoR / Decorator implementations. In the remaining cases, the relevant master object (e.g. Decorator, Handler) is modeled as a *global anonymous object*, thus treating the dataflow in the *field-based* way.
- *Array-element-insensitive* - accesses to different array/collection elements are not distinguished.

Such implementation is build on the basis of empirical observations, following thorough examination of tens of occurrences taken from the repositories mentioned in Sec. 3.1 and experimenting with a tool prototype that delivered a good accuracy and response time. Experiments with the first versions of DPJF showed that more powerful points-to analyses (for example, context-sensitive or flow-sensitive analyses) incur significant additional response time without contributing to the accuracy.

Points-to sets. At the object level, the points-to set of a variable V is computed by monitoring the corresponding memory location. At the program element level, all paths leading into V are examined in the corresponding dataflow graph.

For a variable Var, $pointsTo(Var) = \{Object_1...Object_n\}$ such that Var is dataflow reachable from any $Object_i...Object_n\}$. If there is no Obj such that $Obj \rightarrowtail Var$, it is assumed that some anonymous object of the type T is pointed to by Var.

8.2. Improving Recall

The recall-improving techniques provided in the subsections of this section accept the false negatives mentioned in the State of Art chapter, Sec. 3.4.

Relevant role players reside in missing code. DPJF treats missing code as follows:

(1) Automatically includes all available external dependencies (other projects, widely used bytecode libraries like Java Runtime library) into the analysis. Thus the candidate from Fig. 3.4.16 is accepted.
   - This approach might not be appropriate if repositories which represent frequently used libraries by themselves - (e.g. Java IO which is a part of the Java runtime library) are analyzed. An entity in the analyzed code and in the attached bytecode can possess the same name, leading to conflicts. This issue is being investigated currently.

(2) Role players that are not found even in external bytecode are treated by distinguishing optional roles from mandatory roles. DPJF tolerates missing players for optional roles but requires mandatory ones to be played by at least one program element.

Too strong constraints on attribute values. No restrictions are placed on visibility qualifiers (private, protected, public) in DPJF.

Insufficient treatment of transitivity. DPJF supports the transitivity of the following relations: subtyping, control flow reachability and data flow reachability. For the sake of efficiency, the number of intermediate entities (e.g. intermediate methods in indirect method invocations) is limited; see Section 8.3 for more details.

8.3. Improving Response Time

DPJF uses the following efficiency-improving techniques (described in the subsections of this section):

(1) Improve the efficiency of structural analyses by divide structural analyses into smaller subqueries and apply various optimization techniques. For example, the fastest queries (e.g. dealing with ownership or subtyping) are executed first and put into the cache for further reuse.
(2) Improve the efficiency of behavioral analyses. These are performed only on demand. Dataflow scopes are limited on the basis of empirically validated heuristics.
(3) Compute some core constraints efficienty.

Improving the efficiency of structural analyses. In early DPJF versions all possible combinations of program elements that play mandatory roles in sought patterns were enumerated. This approach appears to be extremely inefficient; enforcing of transitivity aggravated further the situation. The following empirical observations motivated the development of efficiency-improving techniques: [2]

(1) Limited transitivity.
    (a) The number of intermediate classes in the **transitive subtyping relation** does not exceed 2. In addition, certain top elements of subclassing chains such as Java core classes (like *java.Object*) do not play relevant roles in design patterns and are therefore discarded.
    (b) **The method invocation relation** has at most one private intermediate method I.
(2) Direct and indirect calls do not coexist. A method invokes pattern-relevant method calls either directly or indirectly, *but not in both ways.* For example, if a forwarder method $Forwarder$ invokes a forwarding call directly, it is unlikely that $Forwarder$ invokes another forwarding call indirectly.
(3) Repeating queries. For example, subtyping and variable / method ownership appear as subqueries in many queries so their results can be reused.
(4) Structurally similar patterns. Occurrences of some structurally similar patterns are characterized by simple and fast structural queries that are not satisfied on occurrences of other patterns.
    - As an example, a class *Handler* owns a field *ParentField* of the type *Component* (which is a superclass of *Handler*). In addition, some method *Handler.M*(...) overrides a method

[2]These techniques might not be applied when data - and control flow graphs are takes from third-party libraries.

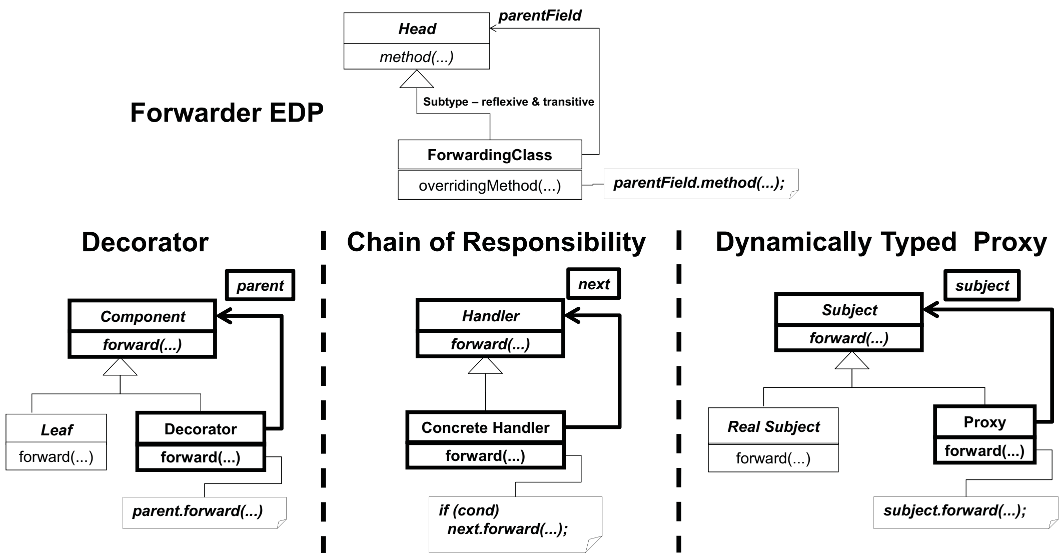


FIGURE 8.3.1. The forward-to-parent similarity group. Thick lines describe the substructures within pattern occurrences that match the EDP.

$Component.M_1(...)$. It is very likely that these program elements play the corresponding roles in a Decorator, CoR or Dynamically Typed Proxy occurrence. These elements form the Forwarder elemental design pattern (EDP)[3] illustrated in Figure 8.3.1. Throughout this thesis, such EDPs are referred to as *similarity group EDPs*. Additional structural and behavioral checks are needed to discriminate patterns within the same similarity group. For example, the Forwarder EDP characterizes the patterns belonging to the *forward-to-parent similarity group*. More details about similarity groups are provided in Appendix A.

Given all of the above factors, the following efficiency-improving techniques are suggested:

(1) Restricted transitivity. Compute the "nearby-transitive" subtyping relation (with at most 2 intermediate classes) omitting the top-level system classes like *java.Object*. The "nearby-transitive" method invocation relationship has at most 1 intermediate method.
(2) Prioritized transitivity. First, one checks whether a method M invokes some pattern-relevant method call directly. If no such call exists, one searches whether M invokes pattern-relevant calls through an intermediate method.

[3] The terminology of Smith and Stotts [35] is used

(3) Caching. Subtyping and variable / method ownership are precomputed and put into the cache after the tool is loaded. The occurrences of EDPs for a group of structurally similar patterns G are pre-computed and cached before the tool detects patterns in G.
(4) Detection order. Assume that occurrences of a certain EDP are generated before detecting patterns in a similarity group G. If these occurrences are used when detecting patterns in other similarity group H, patterns in G must be detected before detecting patterns in H.

Improving the efficiency of behavioral analyses.
Control flow.

Simplified interprocedural control flow. The set of called methods for a method invocation mc consists of the method M statically referred to by mc and of the methods that override M (Class Hierarchy Analysis, see Tip and Palsberg [66]). This algorithm delivers a satisfactory precision. If in the future the precision will not satisfy a user, a call graph could be further refined using points-to information.

Simplified exploration of transitivity. DPJF uses the observation of [64, 73] that in a collaborational pattern (except for Visitor) the receiver variables of all relevant method invocations point to the master object. Additional experiments conducted by Binun et al. [11] confirmed this observation and stated that in these patterns intermediate methods (if exist) are private and are invoked on the *implicit this* receiver. An example is an intermediate method *private int read*1(...) from a CoR occurrence in Figure 3.4.12a.

This observation allows to avoid a complex (possibly transitive) computation of the points-to set of the receiver variable. One needs to check whether an intermediate method is private and the relevant invocation is sent to *this* (explicitly or implicitly).

Data flow.

Limited scope. A caller method M invokes a relevant callee method m on the object stored in a field F. The scope of the dataflow $F \rightarrowtail receiver(mc)$ might be formed not only by M but also by the method(s) $M_1 ... M_k$ directly invoking M.

Therefore, first the dataflow $F \rightarrowtail_M receiver(mc)$ is verified. If this check fails, one checks whether $F \rightarrowtail_{M_1 ... M_k} receiver(mc)$.

Replace dataflow by structural approximations. In certain scenarios the dataflow analyses can be replaced by empirically validated approximations based on structural analyses. According to the observations of the author of this thesis, these scenarios are quite typical and cover approximately 85-90% of the dataflow-relevant situations, thus improving the response time of DPJF. The scenarios and the corresponding simplifications are provided below:

- **Receiver expressions of relevant method calls refer to pattern-relevant fields** and therefore the dataflow from the relevant field(s) to the receiver variables of the relevant method calls should not be actually computed. If the receiver expression is a getter method invocation then this case can be processed in a similar way.

 – For example, in the scenario in Figure 3.4.2b the relevant method calls are done directly on the fields $myFE1, myFE2$ that serve as the receiver variables.

- **Dataflow from a collection field to a scalar variable**: Dataflow from collection-typed fields into scalar-typed variables ($fListeners \rightarrowtail listener$ in Figure 3.4.5a) is easily processed using structural simplifications. Assignment statements (e.g. $listener := l$) and element retrievals (e.g. $l := fListeners.get()$) are collected within the body of the corresponding caller method.

Efficient implementations of some core constraints.

Update Field. The goal is to check the existence of a (public and non-construction) method M that forms the scope of the dataflow into a pattern-relevant field F in a decoupler pattern. Instead of applying full-scaled dataflow analyses, DPJF enforces simple structural checks that approximate dataflow analyses, yielding a high precision.

The implementation is divided into two cases:

F **is scalar-typed**. DPJF checks whether some public and non-construction method M contains the assignment statement $F := expr$.

F **is collection-typed**. The contents of F are usually modified by adding or deleting elements from a collection. The situation is complicated by the fact that the source code of the runtime libraries implementing collection logic are not usually attached to the analyzed project. Denote by O the object owning F and by $DepObjType$ is the type of dependent objects pointed to by F.

DPJF performs two checks:

(1) There exists a method $O.U(..., DepObjType\ v, ...)$ that contains a statement of the form $F.add(v)$

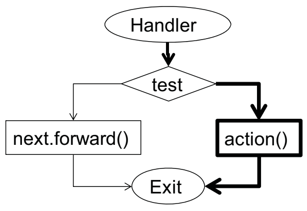
(2) There exists a caller method C that

 (a) forms the scope of the dataflow from F to a var. $DepObjType\ l$

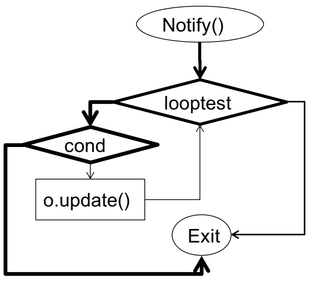
 (b) C invokes a callee method callee in a loop in a statement of the form $l.callee()$

For example, in Figure 3.4.5a U stands for $addDrawingChangeListener(DrawingChangeListener\ listener)$ that updates the relevant field $fListeners$ in the statement $listeners.addElement(listener)$. The caller method $figureInvalidated(...)$ invokes the callee method in a statement $listener.drawingInvalidated(...)$ where $fListeners \rightarrowtail listener$.

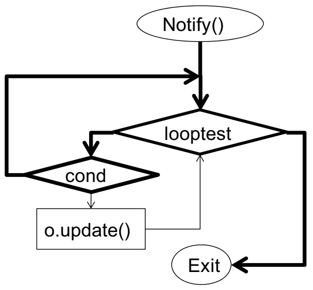
Partial / Full Coverage. The Partial / Full Coverage constraints (Section 7.3) are checked by traversing control flow relations of caller methods, without computing points-to sets. This is done because analyzing control flow graphs takes less time that building points-to sets. The goal is to find a



(A) Scalar method invocation



(B) Breaking an iteration



(C) Omitting an iteration

FIGURE 8.3.2. Applying the Full / Partial coverage constraints. Pattern-relevant paths that represent diverting executions are drawn using the bold line and there is dataflow from relevant field(s) to the receiver variable(s) of the relevant method call(s).

pattern-relevant (Sec. 2.5) path that represents a diverting execution (Sec. 6.2.1) and reflects the application logic typically implemented by a pattern.

The relevant scenarios fall into several categories illustrated by the examples provided below:

- Figure 8.3.2a illustrates the case when the relevant field (here, *next*) is scalar-typed. The relevant method call is $next.forward(...)$. This scenario is typical when distinguishing between CoR and Decorator implementations. A diverting execution is represented by a simple path that does not contain relevant method call statement(s) (e.g. $next.forward(...)$).
- Figure 8.3.2b and Figure 8.3.2c illustrate the cases when the relevant field V is collection-typed. This scenario is typical for false Observer and Composite candidates since the relevant method (e.g. *update*(...)) is not invoked on some object pointed to by a relevant field.
  - Figure 8.3.2b illustrates the case when an iteration over dependent objects is broken. A diverting execution is represented by a simple path that does not contain the statement standing for a relevant method call *o.update*(...)).
  - Figure 8.3.2c illustrates the case when a certain iteration over dependent objects is omitted[4]. A diverting execution is represented by a path that goes through a simple cycle that does not contain relevant method call(s) (e.g. *o.update*(...)).

[4]Usually this is done using the *contnue* operator.

Note that there yet one category of pattern-relevant paths that does not form proper diverting executions. Namely, if a pattern-relevant method call $f.callee()$ resides within a branch of an if-statement $if(f! = null)$ then the path corresponding to the else-branch does not represent application logic (and cannot form a diverting path) since it contains “cleanup” actions that are done by an attempt to perform an invocation of the null pointer.

8.4. Combining of static and dynamic analyses in constraint implementations

Section 8.1 shows how to implement the basic relations and functions using static and dynamic analyses. Therefore it is *possible* to implement each core constraint using dynamic analyses in addition to static analyses, aiming at improving the precision. The constraints can be divided into two groups:

(1) *The whole* behavior is relevant. This group includes the following constraints: Exclusive Dataflow Source&Target, Same Operation, Sibling Creation, State Propagation, Uniquely Created Product Type and Total Coverage.
  - For example, in the Total Coverage constraint each execution must be examined so the whole control- and data flow is important.

(2) *A part of* the whole behavior is relevant. This group includes the following constraints: Field Update, Partial Coverage and Control/Data Flow Order. These constraints can be verified by finding *some execution* that possesses certain properties.
  - In the Field Update constraint the relevant field is required to be updated in some execution. In the Partial Coverage constraint relevant method calls must not be launched in some execution. In the Control / Data Flow order interleaved updates of a certain field and invocations of a certain method are caused by some execution.

The constraints from first group are more suitable for static level implementations since static analyses provide conservative approximation of the whole program behavior.

This thesis suggests static-level implementations also for the constraints from the second group. These constraints are: Field Update and Partial Coverage. The Field Update constraint is implemented by building data flow for relevant construction methods. The Partial Coverage constraint is implemented by examining the control flow graphs for relevant methods. The evaluation shows that these implementations are efficient and accurate; therefore Field Update and Partial Coverage are more suitable for static-level implementations.

The only constraint that is more suitable for dynamic-level implementation is the Control/Data Flow Order constraint. Its current specification

	EDFS &EDFT	Field Update	Same Operation	Sibling Creation	State Propagation	CF-DF Order	Uniq.Created Prod.Type	Total Coverage	Partial Coverage
Relevant Behavior	whole	part	whole	whole	whole	part	whole	whole	part
Suitable For	static	static	static	static	static	dynamic	static	static	static

FIGURE 8.4.1. Implementing constraints

relies on the existence of a setter method that updates the relevant field. However, the recent observations revealed implementation variants where the relevant field is modified by update statements that are not within seetter methods. Implementing such variants at the static level requires the (potentially expensive) flow-sensitive dataflow analysis. On the other hand, at the dynamic level just finding an execution that causes interleaving updates of a certain field and invocations of a certain method suffices. Practical observations show that such executions are not usually formed by conditional statements and therefore cover all relevant behavior.

Figure 8.4.1 describes how the core constraints can be implemented at both levels. Constraints are provided in columns. For each constraint, the values in the first row shows the group a given constraint belongs for (whole / part of behavior is relevant). The value in the second row shows the analysis suitable for implementing a given constraint (static/dynamic).

Implementing constraints at the dynamic level remains a part of the future work.

8.5. The organization of DPJF

Implementation of DPJF in a very short time frame was enabled by the availability of JTransformer [45], an easy to use development environment for analyses and transformations of Java source code. JTransformer deals with the parsing of source and byte-code, resolution of project dependencies and creation of an internal representation of the entire code in Prolog. Analyses can be expressed at a very high level of abstraction as Prolog clauses. DPJF is implemented as a set of SWI-Prolog *detection modules*, each dedicated to the detection of candidates for a group of structurally similar implementations. Figure 8.5.1 associates the detection modules with patterns recognized by them.

Some features are still not implemented. Note that Figure 8.5.1 reflects the conceptual level of DPJF, not the modules that are actually implemented. Currently, DPJF supports detecting only 5 patterns: Decorator, CoR, Proxy, Observer and Composite so only the modules *DetectForwarders*, *DetectSiblings* and *DetectWithUpdatedCollectionField* are implemented. The ovals standing for the actually implemented modules in Fig.8.5.1 are wrapped by thick lines.

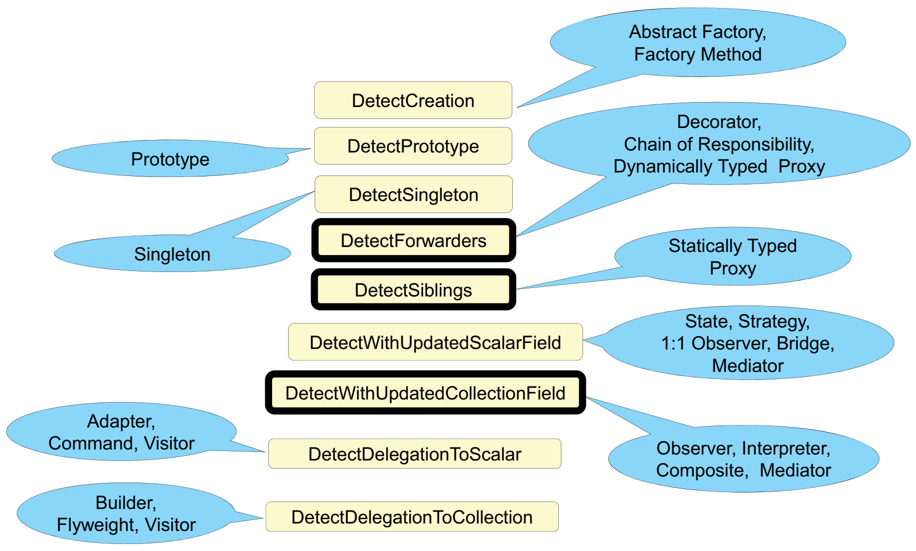


FIGURE 8.5.1. Detection modules supported by DPJF

Also, only the constraints that are required in the specifications of the supported patterns are implemented. That is, Exclusive Data Flow Target, Control/Data Flow Ordering and Uniquely Created Product Type constraints are still not implemented. Implementing of remaining modules and constraints is considered as a part of the future work.

The execution order of detection modules (i.e. workflow) is described using the activity diagram from Figure 8.5.2. It is based on the observation that if a detection module DM_1 generates and caches similarity group EDPs that are reused in another detection module DM_2, then DM_1 is executed before DM_2. For example, the EDP "Create" consists of the method *Creator* that creates object(s) of the type *Product*. This EDP is used first in the module *DetectCreation* that recognizes Abstract Factory and Factory Method. It is later reused in the module $DetectForwarders$ to model possible creations of Real Subject objects in Dynamically Typed Proxy.

Each detection module performs the following steps (described in the subsections):

(1) Find all program elements that play the roles in the pattern implementations within a given similarity group
(2) Aggregate role assignments to candidates.
(3) Apply the core constraints to filter out false positives and properly classify the candidates.

The second step is generic, the others have specific implementations in each detection module. All steps are illustrated on the example of the module $DetectForwarders$ that recognizes candidates in the forwarder-to-parent group.

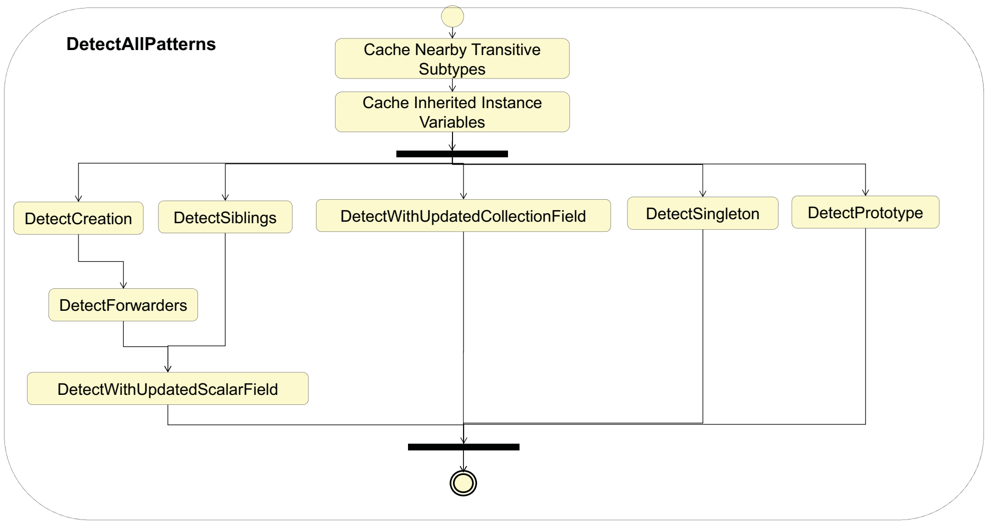
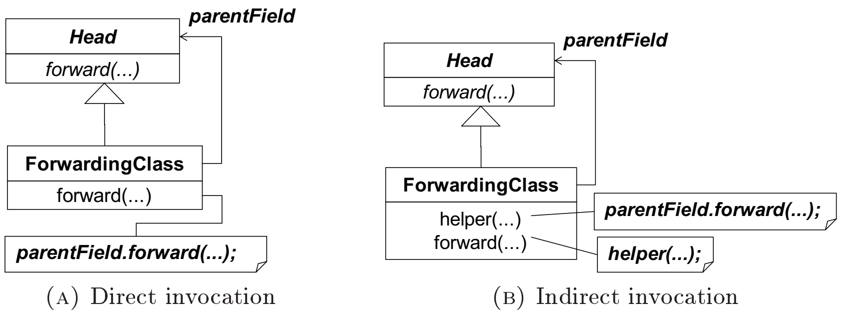

DetectAllPatterns
Cache Nearby Transitive Subtypes
Cache Inherited Instance Variables
DetectCreation
DetectSiblings
DetectWithUpdatedCollectionField
DetectSingleton
DetectPrototype
DetectForwarders
DetectWithUpdatedScalarField

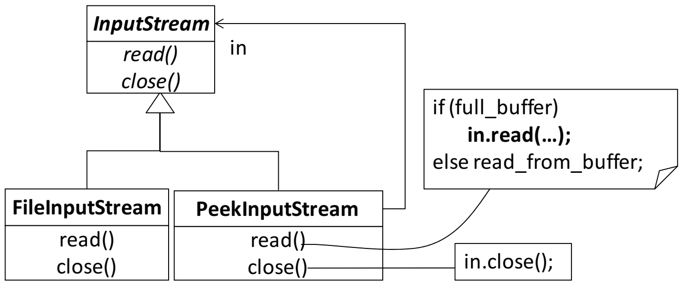
FIGURE 8.5.2. The main DPJF module



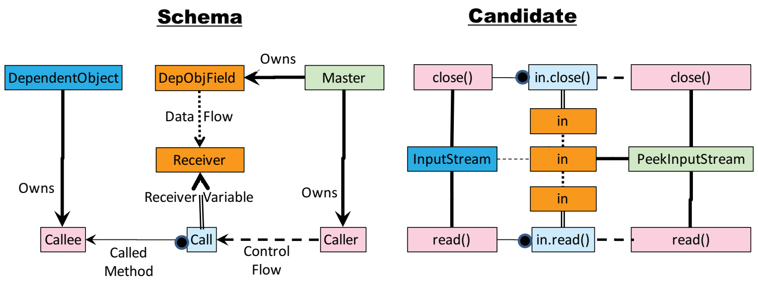
(A) Direct invocation (B) Indirect invocation

FIGURE 8.5.3. Transitivity check

Find role assignments. The analysis starts by computing the subtype and containment relation. Then candidates for the similarity group EDP (Forwarder, Fig. 8.3.1) are computed. They are stored as tuples of the form <*ForwardingClass*, *Head*, *OverridingMethod*, *Method*, *ParentField*>. Each tuple element represents the assumed player of one role in the EDP. Next, program elements that play additional roles are identified. This is done by checking for each candidate how the *OverridingMethod* player invokes the *Method* player. If the invocation is direct (as depicted in Figure 8.5.3a) the tuple is kept unmodified. Otherwise, the element playing the role of *IntermediateMethod* that invokes *Method* (see Figure 8.5.3b) is identified and the candidate is extended by the role assignment for *IntermediateMethod*.



(A) An application excerpt from Java IO 1.4



(B) The corresponding candidate

FIGURE 8.5.4. Creating candidates

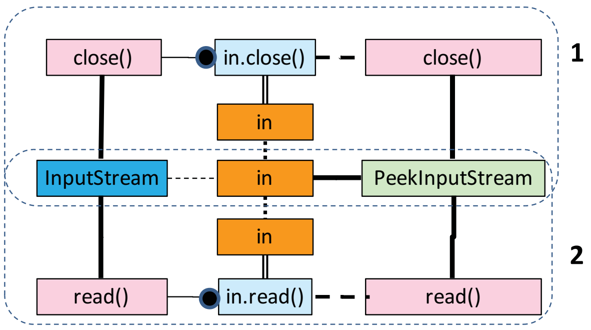
Aggregate role assignments to candidates. Given many players of the same role one needs to distinguish players that belong to different candidates and combine players that belong to the same. Formally, a candidate is a connected component from the projection graph induced by the design pattern schema and the role assignments (Sec.7.2). All players that are related belong to the same candidate. An excerpt from Java IO 1.4 provide in Fig. 8.5.4a gives rise to the candidate provided in Fig. 8.5.4b.

Apply Behavioral Constraints. In the final step, false candidates are eliminated and correct ones are categorized as Tree Forwarder, Decorator, Dynamically Typed Proxy or Chain of Responsibility.

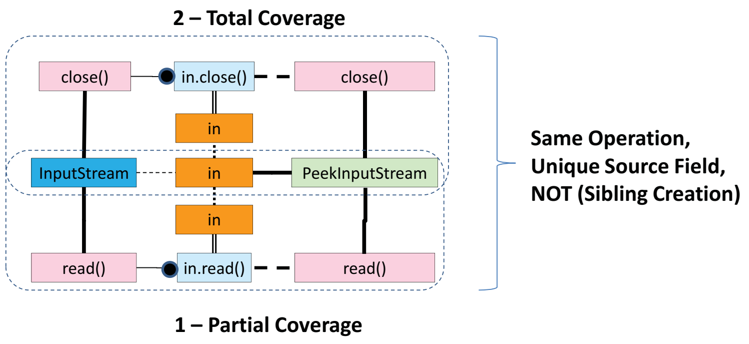
The main problem is that a candidate can include caller methods that represent functionalities of different patterns. For example, in Fig. 8.5.4a the method *PeekInputStream.read()* is a part of a CoR occurrence (forwarding only in some execution) whereas the method *PeekInputStream.close()* is a part of a Decorator forwarder (forwarding in each execution).

First of all, a candidate graph is divided into *collaboration subcandidates* (CSs). Each CS consists of:

- a caller method $Caller$,
- the call statement(s) $call_1 ... call_n$ called by $Caller$,
- the callee methods $Callee_1 ... Callee_m$ referred to by $call_1 ... call_n$



(A) Collaborational subcandidates



(B) Collaborational subcandidates and the constraints satisfied on them

FIGURE 8.5.5. Collaborational subcandidates and constraint application

- the fields(s) $F_1...F_k$ that point to the dependent object(s) on which $call_1...call_n$ are invoked
- the type *Master* that represents master objects owning *Caller*
- the type *Dependent* that represents objects owning $Callee_1...Callee_m$

Note that a CS contains players for each role of the collaborational schema. Fig. 8.5.5a illustrates the CSs belonging to the candidate provided in Fig. 8.5.4b.

The idea behind CSs is motivated by the observation that in collaborational patterns each caller method represents represent a self-contained pieces of functionality. Different constraints can be satisfied on each CS. Therefore CSs on which the same set of constraints are grouped into one candidate.

Now I present the process of constraint application on each CS. First, the CSs on which the Unique Source Field constraint is satisfied are filtered out since they form a Tree Forwarder candidate.

```
algorithm DetectForwarders
input: Candidate candidate;

tuples = decomposeIntoCSs(candidate);
let USFTuples={tuple ∈ tuples: UniqueSourceField(tuple)}
  MSFTuples=tuples / USFTuples;
report formCandidate (MSFTuples) as TreeForwarder;

if ∃ tuple ∈ USFTuples: SiblingCreation(tuple)
then report formCandidate(USFTuples) as Dynamically Typed Proxy;
else
  let TCTuples = {tuple ∈ tuples: TotalCoverage(tuple)}
      PCTuples = {tuple ∈ tuples: PartialCoverage(tuple)}

report formCandidate (TCTuples) as Decorator;
report formCandidate (PCTuples) as Chain of Responsibility;
```

FIGURE 8.5.6. Applying Constraint to detect Forwarder patterns

If the Sibling Creation constraint is satisfied on at least one remaining CS, then all remaining CSs are said to form a Dynamically Typed Proxy candidate. Finally, a Decorator (resp. a CoR) candidate is formed by the CSs on which the Partial (resp. Total) Coverage constraint is satisfied.

The whole constraint application process is illustrated by the pseudocode from Fig. 8.5.6.

.

CHAPTER 9

Evaluation of DPJF

This section explains the evaluation methodology, compares DPJF to four other pattern detection tools and presents the comparison results.

All tools were evaluated on the Decorator, Chain of Responsibility (CoR), Proxy, Observer and Composite patterns. DPJF was compared to four of the five tools evaluated in [**28**, **42**]: Fujaba [**71**], PINOT [**58**], Ptidej [**29**], and SSA [**68**]. DP-Miner [**16**] was not considered because it detects only one pattern (Composite) of the five sought patterns.

Section 9.1 presents the main challenges and approaches when the accuracies of different pattern detection tools are computed. Section 9.2 discusses the evaluation results.

9.1. Computing precision and recall

When the precision and the recall are computed, the reported candidates are compared against the test set occurrences. The following challenges emerge when comparing pattern detection tools:

- Grouping of program elements into candidates / occurrences
- Which program elements participate in the accuracy computation.

The subsections discusses the challenges and shows how these challenges are solved in DPJF.

Accuracy Computation Challenges.

Grouping of program elements into candidates. A candidate/ occurrence is a set of assigments of program elements to roles (Section 7.2). Almost all pattern detection tools mentioned in the thesis ([**2**, **16**, **58**, **12**, **47**, **20**, **71**, **34**, **73**]) do not support role cardinalities; i.e. *at most one program element can be assigned to any role in a given candidate.* The sole exception from this rule is SSA [**68**] which groups pattern-relevant methods around their owning classes, thus allowing method roles to be played by more than one method. DPJF supports cardinality specifications in pattern schemas. In particular, more than one program element is assigned to a role that has the non-negative cardinality.

The main problem is that when DPJF and any other pattern detection tool support the same roles and assign the same program element to these roles, the number of candidates might differ, hindering result comparison.

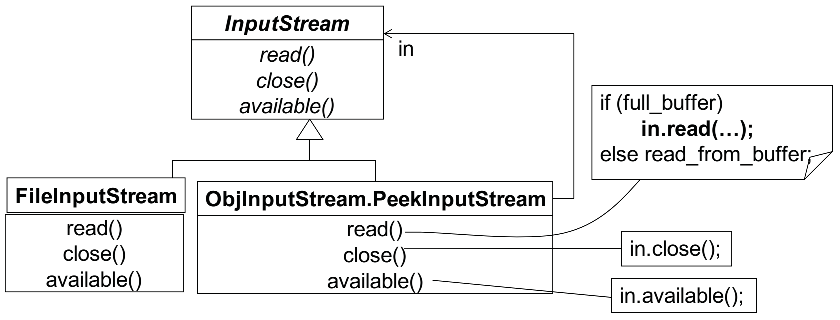


FIGURE 9.1.1. A "mixed" Decorator and CoR candidate

Program elements that participate in the accuracy computation. According to the review of Pettersson et al. [54] that is confirmed by the tool evaluation from Chapter 3, all reviewed pattern detection tools compute precision and recall based on the number of found candidates. If in a given candidate C *some classes playing mandatory roles* are properly recognized, C is accepted. This approach is referred to in this thesis as *the candidate-based accuracy computation.* This is sometimes a too coarse grained metric, since every candidate and occurrence of a given implementation variant consists of role players on different granularity levels (classes, fields, methods, individual statements). Some role players might

- be wrongly identified, leading to *role-level false positives.* For example, within a Decorator candidate certain methods forward conditionally, thus being Chain of Responsibility forwarder methods. However, pattern detection tools that do not employ sufficient behavioral analyses wrongly identify these methods as DECORATOR FORWARDING METHODS. The method $PeekInputStream.read(...)$ from Figure 9.1.1 is a role-level false positive generated by SSA. It is misclassified as a Decorator forwarder, though it forwards conditionally.
- not be identified at all (although the tools search for players of that role), leading to *role-level false negatives.* For example, method role players that invoke relevant method calls indirectly are not recognized by some of the reviewed pattern detection tools. The method $BlockDataInputStream.read(byte[], int, int)$ from Figure 3.4.12a is a method-level false negative since no tool mentioned in this thesis supports transitive method invocations.

That is, in the presence of role-level false positives (or false negatives) a pattern detection tool might accept a candidate that has some "false parts" within or miss some unusually implemented (but proper) parts.

Precision / Recall Computation in DPJF.

Grouping program elements into candidates. For each evaluated pattern detection tool the candidates are recomputed in the same way as DPJF does. That is, for each evaluated tool T there is the module that parses the output of T. Building of the projection graph and computing of pattern candidates is done exactly like in DPJF.

Program elements participating in accuracy computation. In addition to the candidate-based accuracy computation approach this section presents the accuracy computation approach that *takes into account* role-level false positives and false negatives. This approach is a result of elaborating the suggestion of Pettersson et al. [54]. The individual precision and recall are assigned for each candidate C, taking into account role-level false negatives and false positives. This approach is referred to as the *role-based accuracy computation.*

The *individual precision* for C is the number of correctly recognized role players in C divided by the number of role players reported for C. The *individual recall* for C is the number of correctly retrieved role players in C divided by the total number of actual role players in C. When computing scores in this way, only class ane method role players are processed since these are reported by almost all compared pattern detection tools[1]. The precision of a pattern detection tool on a particular program is computed by dividing the sum of individual precisions of all correctly identified candidates by the number of reported candidates. The recall is obtained by dividing the sum of individual recalls of all reported candidates by the number of occurrences in the test set.

Let C_T be a (possibly incomplete and partly wrong) candidate retrieved by tool T and O be the corresponding complete and correct occurrence that an ideal tool would report (if C_T is a false positive then O will be empty). Formally,

- $\mathit{reported}(C_T)$ is the number of role assignments in C reported by T
- $\mathit{detected}(C_T)$ is the number of correct role assignments in C reported by T
- $\mathit{correct}(C_T)$ is the maximum number of correct role assignments in O for the role types supported by T (e.g. if T is not able to report statement-level roles, then only the correct assignment to class, method and field level roles in O are counted).
- $\mathit{prec}(C) = \mathit{detected}(C)/\mathit{reported}(C)$ is the *individual precision* for C_T
- $\mathit{rec}(C) = \mathit{detected}(C)/\mathit{correct}(C)$ is the *individual recall* for C_T

[1]The only exception is Ptidej. It reports only class roles which are compared. Thus Ptidej is slightly favoured, since it is not penalized for not identifying players of method and field roles, whereas tools that support these roles are penalized when they *misrecognize* players.

If T reports N candidates $C_T^1...C_T^N$ for a program that contains M proper occurrences of some pattern P then:

- $precision_T = \left(\sum_{i=1..n} prec(C_T^i)\right)/N$ is the average precision of T for P.
- $recall_T = \left(\sum_{i=1..n} rec(C_T^i)\right)/M$ is the average recall of T for P.

The evaluation results are discussed in Section 9.2.

9.2. Discussion of Results for Precision and Recall

This section presents the precision and the recall computed delivered by the role-based computational approach and compares them to the accuracies delivered by the candidate-based approach. Eventually, the response times are presented.

Results for the Role-Based Computation Approach. For each pattern, the results are shown in one of the tables of Fig.9.2.1. Column headings indicate the analysed project and (in brackets) the number of known occurrences for that project. The “Total” column indicates the performance when all projects are taken as a single big one. Precision and recall are computed as described in Sec.9.1. A bomb indicates that the tool crashed on that project. A precision (resp. recall) value that cannot be computed because there are zero occurrences (resp. candidates) is indicated by “—”.

The values for different projects are aggregated into an *average* per pattern and tool, computed as if all tested projects were a single big one. The average values are shown in the “Total” Column of each table of Fig. 9.2.1. For easier reference and comparison they are summarized (together with their median) in Fig. 9.2.2a, Fig. 9.2.2b and are depicted graphically in Fig. 9.2.2c and Fig. 9.2.2d.

Achieved Precision. Fig. 9.2.2a and 9.2.2c show that the average precision of the previously existing tools is highly dependent on the pattern and benchmark, with medians ranging from 10% for Pinot to 66% for SSA. In contrast, DPJF achieves 100% precision for each evaluated pattern and benchmark.

Obviously, the constraints supported only by DPJF effectively discard all false positives produced in other reviewed tools.

Achieved Recall. Fig. 9.2.2d and Fig. 9.2.1 show that DPJF has the best recall among all evaluated tools. The median of DPJF’s recall is 89%, which is 2 to 4.5 times better than medians of 18% for Ptidej to at most 44% for SSA. The differences are even more pronounced when comparing individual recalls. That is, the recall-improving techniques implemented in DPJF are very effective too. The main influencing factor for the recall is the exploration depth of transitive relations.

The Role-Based vs. the Candidate-Based Accuracy Computation. This section compares the metrics computed in the candidate-based way deviate to the metrics computed in the role-based way (Fig. 9.2.1) and

Decorator	JHD 5.1 (3)				JHD 6 (10)				JavaIO (4)				AWT (9)				ArgoUML (3)				TeamCore (2)				Total (31)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	0	---	0	---	5	59	26	36	4	78	68	72	3	96	33	49	1	100	33	50	1	100	22	35	14	78	31	45
PTIDEJ	6	17	33	22	8	25	20	22	3	33	17	22	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	17	24	12	16
SSA	3	67	67	67	8	60	50	54	2	83	23	35	7	76	53	63	1	100	33	50	0	---	0	---	21	70	44	54
DPJF	3	100	100	100	9	100	90	95	4	100	93	96	8	100	89	94	3	100	100	100	1	100	50	67	28	100	89	94

(A) Results for Decorator (Fujaba is not shown since it does not detect the pattern)

Chain of Responsibility	JHD 5.1 (0)				JHD 6 (3)				JavaIO (2)				AWT (4)				ArgoUML (0)				TeamCore (3)				Total (12)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	0	---	---	---	1	0	0	---	2	25	35	29	4	55	54	54	1	0	---	---	0	---	0	---	8	34	24	28
PTIDEJ	2	0	---	---	4	25	33	29	3	67	100	80	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	9	33	18	23
DPJF	0	---	---	---	3	100	100	100	2	100	100	100	4	100	100	100	0	---	---	---	2	100	67	80	11	100	92	96

(B) Results for Chain of Responsibility (Fujaba and SSA are shown since they do not detect the pattern)

Proxy	JHD 5.1 (2)				JHD 6 (2)				JavaIO (7)				AWT (6)				ArgoUML (9)				TeamCore (0)				Total (26)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
PINOT	2	0	0	---	11	9	50	15	3	67	23	34	10	10	17	13	10	10	11	11	0	---	---	---	36	14	18	16
PTIDEJ	1	0	0	---	1	0	0	---	0	---	0	---	💣	💣	💣	💣	💣	💣	💣	💣	0	---	---	---	2	0	0	---
SSA	0	---	0	---	0	---	0	---	6	67	33	44	2	100	33	50	2	100	22	36	0	---	---	---	10	80	24	37
DPJF	2	100	100	100	2	100	100	100	7	100	100	100	6	100	100	100	8	100	89	94	0	---	---	---	25	100	96	98

(C) Results for Proxy (Fujaba is not shown since it does not detect the pattern)

Observer	JHD 5.1 (2)				JHD 6 (6)				JavaIO (0)				AWT (2)				ArgoUML (5)				TeamCore (2)				Total (17)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
Fujaba	19	11	100	19	21	14	50	22	💣	💣	💣	💣	💣	💣	💣	💣	23	13	60	21	14	0	0	---	77	10	47	17
PINOT	1	0	0	---	2	100	33	50	6	0	---	---	9	2	10	4	6	0	0	---	0	---	0	---	24	9	13	11
PTIDEJ	17	12	100	21	3	33	17	22	0	---	---	---	💣	💣	💣	💣	💣	💣	💣	💣	0	---	0	---	20	15	18	16
SSA	3	33	50	40	7	76	100	87	0	---	---	---	2	40	40	40	1	100	20	33	0	---	0	---	13	63	52	57
DPJF	1	100	50	67	6	100	100	100	0	---	---	---	1	100	50	67	3	100	60	75	0	---	0	---	11	100	65	79

(D) Results for Observer

Composite	JHD 5.1 (1)				JHD 6 (1)				JavaIO (0)				AWT (2)				ArgoUML (2)				TeamCore (0)				Total (6)			
	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision %	Recall %	F-mean %	Reported	Precision	Recall	F-mean %
Fujaba	1	100	100	100	13	0	0	---	💣	💣	💣	💣	💣	💣	💣	💣	2	50	50	50	7	0	---	---	23	9	33	14
PINOT	1	0	0	---	1	0	0	---	2	0	---	---	3	33	50	40	10	20	100	33	0	---	---	---	17	18	50	26
PTIDEJ	3	33	100	50	5	20	100	33	0	---	---	---	💣	💣	💣	💣	💣	💣	💣	💣	2	0	---	---	10	20	33	25
SSA	1	60	60	60	2	22	100	35	0	---	---	---	0	---	0	---	1	100	50	67	0	---	---	---	4	51	43	47
DPJF	1	100	100	100	1	100	100	100	0	---	---	---	1	100	50	67	1	100	50	67	0	---	---	---	4	100	67	80

(E) Results for Composite

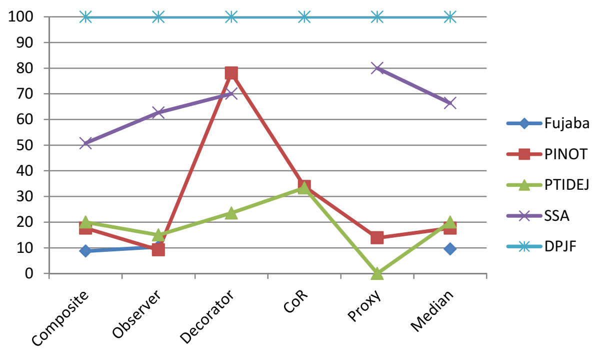
FIGURE 9.2.1. Evaluation results per pattern.

Precision	Composite	Observer	Decorator	CoR	Proxy	Median
Fujaba	9	10	X	X	X	10
PINOT	18	9	78	34	14	18
PTIDEJ	20	15	24	33	0	20
SSA	51	63	70	X	80	66
DPJF	100	100	100	100	100	100

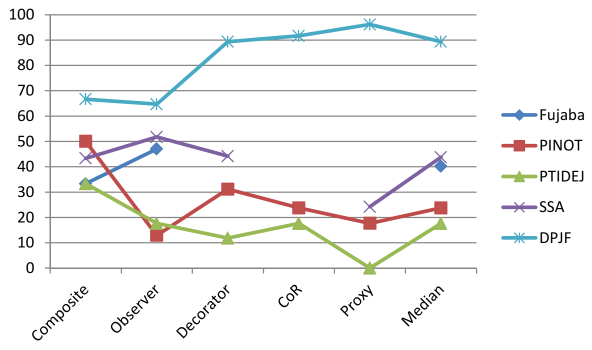
(A) Precision

Recall	Composite	Observer	Decorator	CoR	Proxy	Median
Fujaba	33	47	X	X	X	40
PINOT	50	13	31	24	18	24
PTIDEJ	33	18	12	18	0	18
SSA	43	52	44	X	24	44
DPJF	67	65	89	92	96	89

(B) Recall



(C) : Precision (graph for Fig. 9.2.2a)



(D) Recall (graph for Fig. 9.2.2b)

FIGURE 9.2.2. Precision and recall on all benchmark projects. The inability of a tool to detect a pattern is indicated by X in the table and a hole in the graph

discuss the results. The comparison results are displayed as bar diagrams and are presented in Figure 9.2.3.

The confidence scores computed according to the role-based (respectively, candidate-based) approaches are compared using bar diagrams and are presented in Figures 9.2.3 and Fig. If these approaches deliver different confidence scores for a pattern P and for a benchmark B, two bar diagrams are provided. They compare the precision (respectively, the recall) delivered by each tool that is executed on B to detect P and are placed into a separate subfigure that is contained in Figures 9.2.3, 9.2.4 and 9.2.5.

In each such subfigure the blue bars stand for the accuracies delivered by the role-based approach (denoted by *Precision(R)* and *Recall(R)* respectively). The red bars stand for the accuracies delivered by the candidate-based approach (denoted by *Precision(C)* and *Recall(C)* , respectively).

The differences are highly dependent on the pattern and benchmark; for some patterns and benchmarks there are no differences at all. Therefore it was decided not to compare the average results over all benchmarks per pattern but to present the (existing) differences for individual patterns and benchmarks.

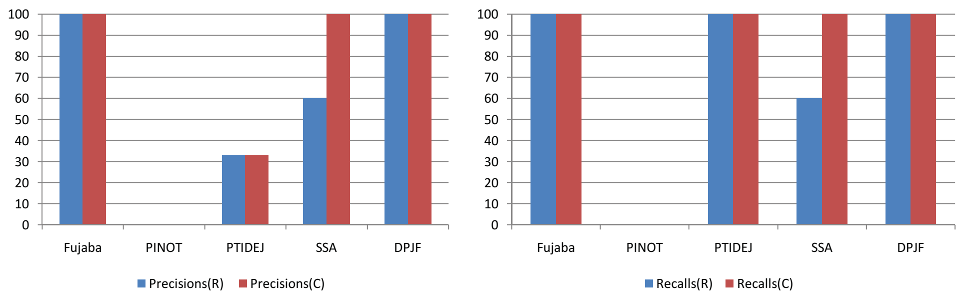
The scores computed in the role-based way are slightly smaller. The differences vary from 4% when comparing the precision for the Decorator pattern in JHotDraw 6.0 up to 40% for the Composite pattern in JHotDraw 5.1. This is because method-level false positives within a given candidate decrease the scores of this candidate when the role-based computation is applied. In contrast, the candidate-based computation approach does not penalize the tools for misrecognizing some method role players.

Achieved Speed. All speed measurements were carried out on a Dell Precision 370 desktop computer equiped with a 3,6 GHz quadcore CPU and 2 GB of RAM running Windows XP SP 3, Eclipse 3.7, SWI-Prolog 5.11.29 and JTransformer 2.9.3. The response times were measured automatically for the tools downloaded with the source code (DPJF, Ptidej and SSA) and manually for PINOT and Fujaba. Each tool was executed ten times on each benchmark and the medians are reported.

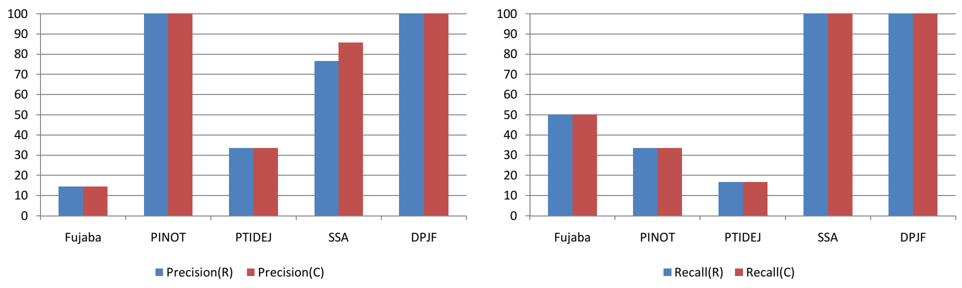
Total run-time. Fig. 9.2.6a and 9.2.6d show the total time that a tool needs for one complete run. This includes initialization (parsing of the project, creating of the internal representation) and query time (searching for pattern candidates). The figures show that PINOT and SSA are the two fastest tools. However, PINOT detected 17 patterns[2] while SSA and all the other tools where selectively run on just 5 patterns. Thus, when running for the exactly same set of patterns, PINOT is presumably the fastest tool on all the benchmarks.

DPJF comes third, clearly outperforming Fujaba and Ptidej. This is remarkable, given its significantly better precision and recall. This might be

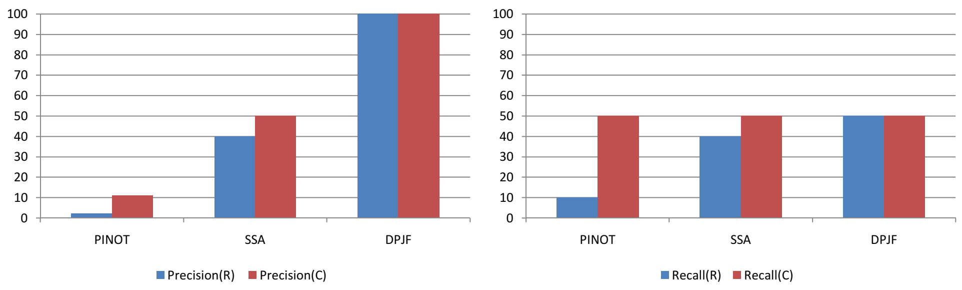
[2]PINOT cannot be run selectively just for certain patterns.



(A) The Composite Pattern (JHotDraw 5.1)



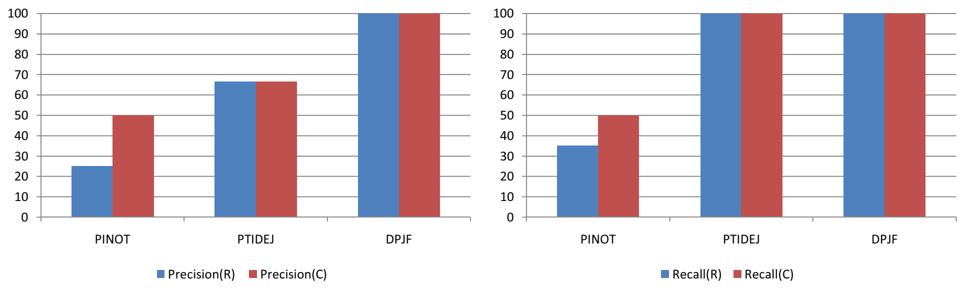
(B) The Observer pattern (JHotDraw 6.0)



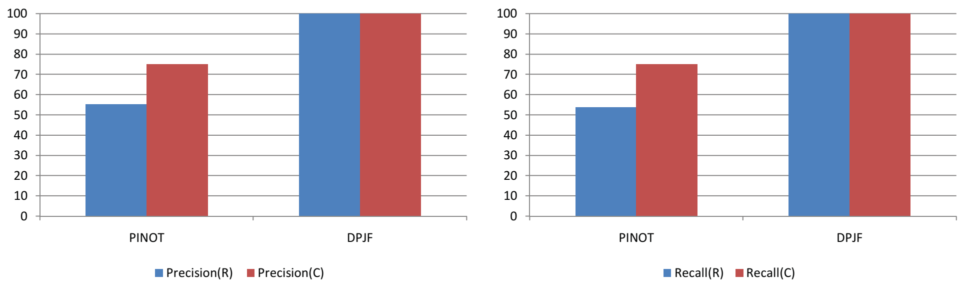
(C) The Observer pattern (AWT 1.14)

FIGURE 9.2.3. Candidate-Based vs. Role-Based Accuracy Computation (Excerpt 1).

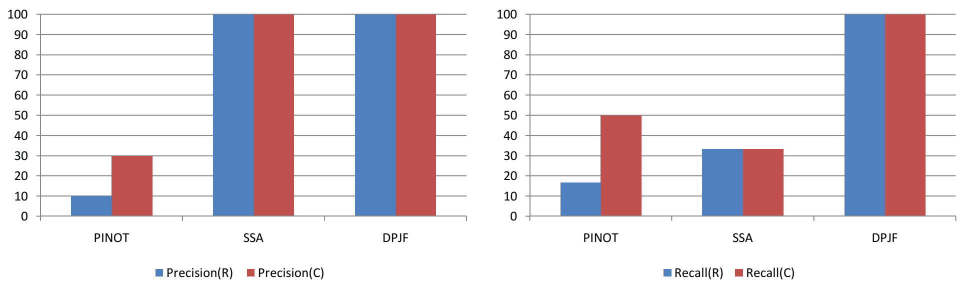
an indication of the immaturity of the base technologies employed in the reviewed tools: Fujaba uses graph transformation and Ptidej uses explanation-based constraint solving. Both technologies are much younger than Prolog and are apparently still lacking compilers and run-time systems that could compete with top Prolog compilers. If Fujaba and Ptidej are disregarded, the third position of DPJF after Pinot and SSA *appears* to confirm the common wisdom that improving precision and recall adversely affects speed.



(A) The CoR pattern (JavaIO 1.4)



(B) The CoR pattern (AWT 1.14)

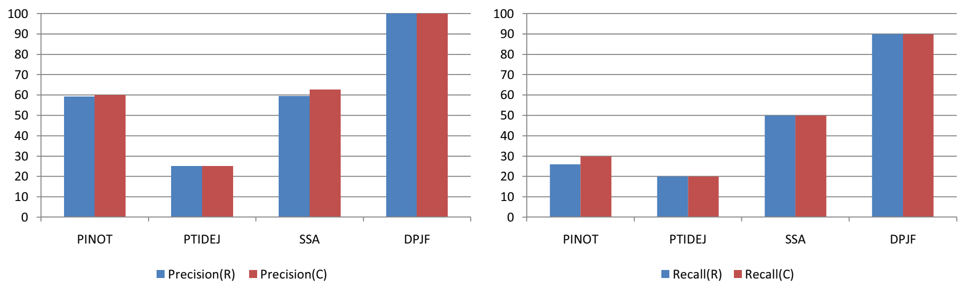


(C) The Proxy pattern (AWT 1.14)

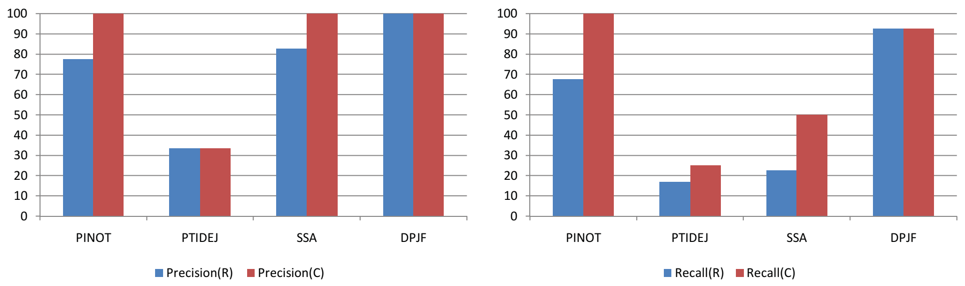
FIGURE 9.2.4. Candidate-Based vs. Role-Based Accuracy Computation (Excerpt 2).

Startup time. The goal was to investigate whether the above result also holds when differentiating the speed of initialisation and of the real analysis. Fig. 9.2.6b shows the initialization time of DPJF, SSA and Ptidej[3]. Comparison with Fig. 9.2.6a reveals that most of SSA's and DPJF's run-time is spent on initialization. This observation does not hold for Ptidej. Comparing

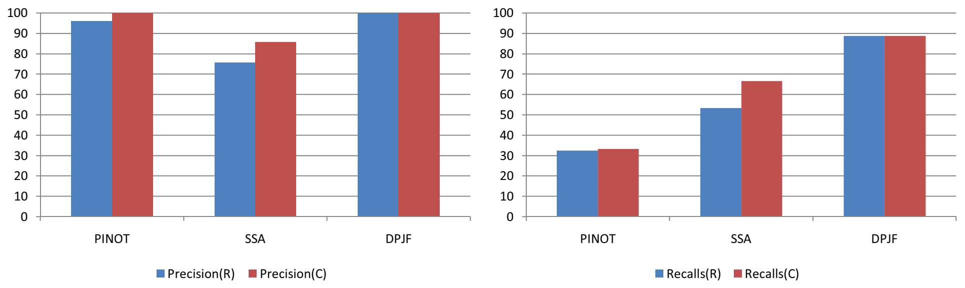
[3]Great thanks to the authors of SSA for providing the source code and the necessary assistance.



(A) The Decorator pattern (JHotDraw 6.0)



(B) The Decorator pattern (Java IO 1.4)



(C) The Decorator pattern (AWT 1.14)

FIGURE 9.2.5. Candidate-Based vs. Role-Based Accuracy Computation (Excerpt 3).

SSA and DPJF, the startup time of DPJF is clearly slower since DPJF *generates* the full representation of the underlying program. This is a problem that DPJF inherits from the current JTransformer version. However, when DPJF is shut down, the representation is cached on disk so that every future restart only *reloads* the representation and is thus significantly faster (compare the rows “DPJF Reload” and “DPJF Generate”). Still, the restart time of DPJF is slower by a factor of 3 to 20 than the startup time of SSA. This is because SSA is a specialized tool that only creates an internal model for the program elements that it needs, whereas JTransformer always creates

	Load and query all evaluated patterns					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Fujaba	60,5	179,9	💣	💣	600,2	50,2
Pinot (all)	**1,0**	**6,1**	**2,1**	17,3	**55,1**	4,0
Ptidej	40,0	99,0	9,1	💣	💣	16,0
SSA	1,5	8,3	3,0	**15,7**	35,2	**0,9**
DPJF	5,6	19,7	6,3	54,1	76,1	6,9

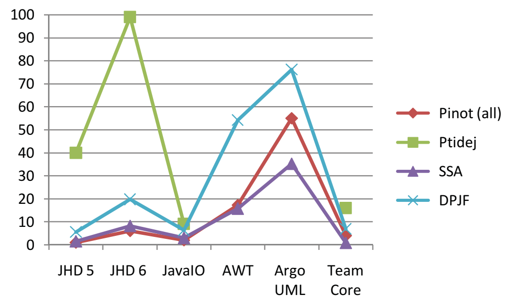
(A) Time for initialisation and analysis

	Preprocessing (seconds)					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej Load	2,235	**4,201**	2,649	💣	💣	5,984
SSA Load	**0,795**	**4,201**	**0,951**	**2,126**	**9,94**	**0,73**
DPJF Reload	4,002	14,05	5,069	39,96	55,06	5,984
DPJF Generate	35	150	49	390	550	60

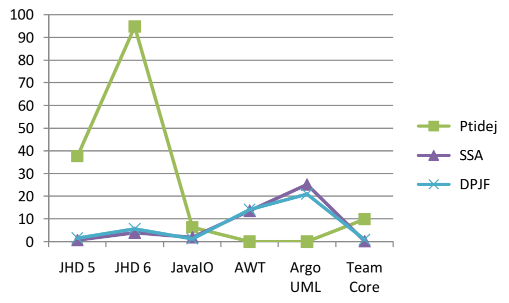
(B) Initialization time

	Query all evaluated patterns					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej	37,7	94,8	6,4	💣	💣	10,0
SSA	**0,7**	**4,1**	2,0	**13,6**	25,3	**0,2**
DPJF	1,6	5,7	**1,2**	14,2	**21,0**	0,9

(C) Pure analysis time



(D) Time for initialisation and analysis



(E) Pure analysis time

FIGURE 9.2.6. Response times of each tool for detecting all evaluated patterns. In the tables, the fastest time per project is red and a bomb indicates that the tool crashed on that project. In the graphs, Fujaba is not shown since it is an order of magnitude slower than most other tools, which would have distorted the graphs. Pure analysis time is indicated only for the tools for which it can be measured separately.

	CoR + Decorator + Proxy						Observer + Composite					
	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core	JHD 5	JHD 6	JavaIO	AWT	Argo UML	Team Core
Ptidej	5,9	24,9	4,1	💣	💣	2,7	31,8	69,9	2,3	💣	💣	7,3
SSA	**0,2**	**1,1**	**1,0**	**3,6**	**5,1**	**0,1**	**0,5**	3,0	1,0	10,0	20,1	**0,1**
DPJF	1,1	3,9	**1,0**	7,0	9,0	0,5	**0,5**	**1,7**	**0,3**	**7,1**	**12,1**	0,4

FIGURE 9.2.7. Query times by pattern group. Shortest time is red.

a full representation of the entire program (including even comments and referenced byte code) since it cannot know real needs of a client application.

Query time. Fig. 9.2.6c and 9.2.6e show the pure query times for DPJF, SSA and Ptidej. The results clearly refute the conjecture that better quality implies lower speed: Obviously, DPJF is competing with SSA for the top position.

Query time per pattern group. Further refining the findings provided in the previous sections, Fig. 9.2.7 compares the query-time of Ptidej, SSA and DPJF when run selectively for individual pattern groups. The numbers indicate the time for detecting all candidates of all patterns in the respective group on a given project. Again, there is no clear winner. For Observer and Composite, DPJF is the fastest almost everywhere except for Eclipse Team Core. When detecting Decorator, CoR and Proxy candidates, SSA is the fastest almost everywhere except for JavaIO (where DPJF is equally fast).

It should be noted that when tools are executed on the same benchmark, the response time of DPJF on the second and subsequent runs is at least 10% lower than the response time on the first run (which is the maximal one among all ten runs, see Appendix B for more details). This might be explained by at least one of the following reasons:

(1) DPJF precomputes and caches the subtyping and ownership relations (and possibly control - and data flow relations) which are reused later.
(2) SWI-Prolog maintains its internal cache where some computation results are stored.

This issue is currently under investigation.

Incremental update. DPJF (thanks to the JTransformer infrastructure) supports *incremental update*: If a file is changed at run-time, only the related part of the factbase is updated, with no noticeable delays. In all but its very first run during an Eclipse session DPJF will therefore incur only the *pure query time* from Fig. 9.2.6c. To the best of the author's knowledge, none of the other tools supports incremental update. They all need to re-parse the entire project for each new analysis, incurring on *each* run the total run-time from Fig. 9.2.6a.

DPJF vs. the Data fusion approach. In the data fusion approach presented in Chapter 4 handling of common witnesses, joint recognition and disagreement appears to be extremely dependent on the particular pattern under study, prohibiting a unified approach that could be automated. DPJF, in contrast, treats all collaborational patterns automatically and uniformly. Treating creational patterns differs only slightly; necessary adaptation can be quickly done.

The accuracy delivered by DPJF is more precise than the one delivered by data fusion (see Fig. 4.7.1). This happens because in DPJF more mature control sets were used. In addition, DPJF does not rely on (possibly wrong) judgements of other tools.

Threats to Validity. The strive for avoiding overfitting by including non-standard projects into the evaluation bears the risk that there are no reference evaluations of these projects yet. So other researchers might disagree with the judgements presented in the thesis. To tackle this threat the results to the public DPB repository of Arcelli, Zanoni and Caracciolo [5] will be made available so that other researchers can assess them.

Crashes also pose a threat to validity. Fujaba crashed when processing AWT 1.3 and Ptidej crashed when processing AWT 1.3 and Argo UML 0.18. It is believed that in both cases the crashes occur due to bugs in the tools. Fujaba delivers the error message ”Cannot parse the files” without providing any details. Ptidej crashes because the Java exceptions *padl.kernel.exception.ModelDeclarationException* and *java.lang.reflection.InvocationTargetException* are unhandled. This information might assist the authors of these tools in bug fixing.

Part 4

Related Work and Conclusions

CHAPTER 10

Related Work

10.1. Efficient Implementations of Behavioral Analyses

Sridharan [62] mentions the following points that characterize behavioral analyses in real program comprehension tasks:

(1) Demand-driven - behavioral analyses are performed only when structural analyses yield a low precision.
(2) Limited analysis scope - the set of relevant methods whose actions consitute the relevant behavior is usually limited.

Given all of the above factors, the following optimizations are suggested to tackle the above mentioned points:

(1) Behavioral analyses are performed only when needed. First try to solve a situation using a fast structural analysis. If the result is imprecise, feed it into a behavioral analysis module.
(2) Refinement-based algorithms. Efficiency can further be increased by starting with a cheap field-based analysis and refining it incrementally towards more and more detailed one until sufficient precision for answering a particular question is achieved. Then the analysis can be stopped, without having to explore all possible paths in a call or flow graph.
(3) On-the fly algorithms. The mutual dependency of points-to analysis precision on call graph precision can be addressed efficiently by on-the-fly construction of the call graph during points-to analysis.

10.2. Design Pattern Detection Approaches

Pattern detection, like any information retrieval task, suffers from false negatives and false positives. These are conflicting issues. Pattern detection tools that apply liberal detection criteria increase recall. On the other side, they decrease precision. In contrast, tools that apply too restrictive detection criteria filter out many false positives, but correct occurrences may be filtered out as well. In addition, the need for speed adversely affects precision as well as recall. Pattern detection tools use the following techniques to improve accuracy:

10.2.1. Improving Precision. Antoniol, Fiutem and Cristoforetti [3] 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 behavioral analyses** ([58, 16]) help to approximate program behavior more reliably thus eliminating false positives. This is an effective but potentially expensive approach.
- **Dynamic analyses** ([75, 74, 47, 31]) 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** ([33, 3, 20]) is based on the observation that certain software metrics (e.g. cohesion, coupling) reflect the specifics of collaborations between entities in patterns. Metric-based fingerprinting is confirmed by empirical studies that in the presence of certain pattern occurrences the values of these metrics reside within certain ranges. These ranges can be found manually ([33, 3]) or using machine learning ([20]). This approach, however, works only for small programs where pattern occurrences do not overlap. During maintenance, many unordered entities and relationships are added. This leads to a lot of overlapping pattern occurrences [48] 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 an occurrence. But some implementation variants might be missed, reducing recall as a consequence.

10.2.2. Improving Recall. Current recall-improving techniques include:

- **Decomposition of patterns** into *elemental design patterns (EDPs)* [36]. EDPs are so small that there are hardly any opportunities for implementation variants. Therefore they are easily and unambiguously recognized by the basic pattern detection techniques. EDPs are used in the following tools: SPQR [36], Fujaba [71] and EDPDetector4Java [4].
- **Using similarity** between the expected and the found structure and behaviour as a criterion (instead of precise matching). For instance, the Similarity Scoring Approach (SSA, [68]) measures similarity between the graph representations of the sought pattern and of the analysed program. Columbus [20] applies machine learning and measures similarity between the metric values of the classes got from training and the real values. Ptidej [31] models the similarity by letting some minor pattern-specifying constraints fail. Fujaba [71] models the similarity by letting some subpatterns that comprise a given pattern to be matched not ideally.

- **Supporting transitivity**. SPQR [36], SSA [68] and Ptidej [31] supports the transitive subtyping relation; SPQR supports the transitive method invocation relation. D-Cubed [64] supports the transitive dataflow reachability relation.
- **Allowing missing players for optional roles** - the players of some roles might be missed. For example, SSA [68] detects several Decorator candidates in Java AWT although these do not include program elements playing the roles CONCRETE DECORATOR or CONCRETE COMPONENT.
- **Relaxed delegation**. Some tools (e.g. Pinot [58]) are restrictive when they require relevant method calls to be executed directly on relevant fields. The occurrences when the pointer to the delegatee is returned by a getter method are missed. To accept such misses, DP-Miner [16] does not check the contents of receiver variables. SSA [68] 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. These can be excluded by checking whether the receiver variable is dataflow reachable from the relevant field.
- **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, [31]). Ptidej, for example, allows LEAF role players in the Composite pattern to be interfaces instead of concrete classes.

10.2.3. Improving Speed. Antoniol, Fiutem and Cristoforetti [3] 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 $O(n^{\mathrm{k}})$ 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. [33] also has the effect of speeding up the detection process by reducing the search space. Dong and Peng [16] and also Tonella et.al. [67] 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 [74] 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 occurrences can be missed since dynamic analyses cannot cover all execution traces.

10.3. Specifying interactions between pattern participants

When discussing pattern specification approaches, this thesis distinguishes *the external approaches* (specifications are written in some high-level language and are accepted as the input by a pattern detection tool) and *the internal approaches* (specifications are “hardwired”, i.e. constitute a part of the design pattern tool implementation). The internal approach is used in all pattern detection tools evaluated in Chapter 3. Among the external approaches, the following ones are mentioned:

- Visual languages. UML `http://www.omg.org/spec/UML/2.4/` is the most frequently used visual formalism to specify pattern implementations. It was used in the specifications of Lauder et al.[46], Wenzel et al. [76] and Kim et al. [41]. In Fujaba (Detten et al.[71]) the decomposition of patterns into simpler patterns is described using a visual language; pattern implementations are modeled as graphs. In the visual language developed by De Lucia et al.[47, 15] pattern participants are modeled by entities (classes, methods) and pattern-relevant collaborations between them are modeled as arcs between entities.
- First-order logic languages are used in Blewitt et al. [12], Zhu et al. [8] and Stencel et al. [64]. De Lucia et al. [15] uses first-order logic enriched with temporal facilities when specifying pattern-relevant collaborations.
- Other ways to model the entities playing roles in pattern-relevant collaborations are: bit vectors (Kaczor et al. [37]) and entity-relationship descriptions (Antoniol et al. [3]).

Note that the above approaches specify only *interactions in pattern occurrences*. Other pattern-relevant aspects such as intent or consequences are addressed more rarely. For example, Kampffmeyer et al. [38] specifies pattern intentions using an ontology that relates design patterns with one another and with associated design problem categories (decoupling, object creations).

10.4. Benchmarking and Tool Comparison

Sim et al. [59] stressed the positive impact of benchmarking on software engineering research by reviewing the most important aspects and challenges of benchmarking. They define a benchmark as ”a test or set of tests used to compare the performance of alternative tools or techniques”. A successful benchmark must meet certain preconditions: a minimum level of maturity of the given research area and the existence of diverse approaches in the field.

Guéhéneuc et al. [32] presented a comparative framework for pattern detection tools. This framework compares certain qualitative aspects such as tool user category (for example, experienced or novice users) or underlying input (for example: is documentation required ? does an input metamodel exist?). Such framework is definitely needed because comparing pattern detection tools is very difficult given that they differ in representations, output formats and implementation techniques.

The benchmark DEEBEE (DEsign pattern Evaluation BEnchmark Environment) developed by Fülöp et al. [25, 26] is used to provide a common platform for reasoning about pattern occurrences and for comparing pattern detection tools based on their results. This benchmark supports several programming languages (Java and C++) and contains results obtained from several pattern detection tools. This benchmark is open to the community and freely available. The benchmark database contains the results of the following pattern detection tools: Columbus (C++), Maisa (C++), and SSA (Java). The tools were evaluated on several ad-hoc examples, on implementation variants taken from the relevant literature and on open source software (Mozilla, NotePad++, JHotDraw, JRefactory and JUnit). Occurrences manually recovered by individual researchers as well as candidates obtained from pattern detection tools can be added to the database and verified by experienced developers. The graphical user-friendly interface provides necessary assistance.

Like DEEBEE, the benchmark DPB http://essere.disco.unimib.it:8080/DPBWeb/ is also used to evaluate pattern occurrences and to compare pattern detection tools based on their results. This benchmark contains evaluated pattern occurrences and candidates obtained from two pattern detection tools written by the developers of this benchmark that employ only structural program analyses. These tools were evaluated on a wide variety of Java projects (simple ones like JHotDraw as well as real-life applications like JRefactory 2.6 and PMD 1.8). Occurrences recovered by individual experts and candidates obtained from pattern detection tools can be added to the database and evaluated using a user-friendly interface.

The benchmark P-Mart http://www.ptidej.net/downloads/pmart/ is a repository that keeps pattern occurrences found in several Java repositories. The identification was done by manually verifying the results of Ptidej. Ptidej was executed on the repositories of various sizes and complexities: simple repositories that contain mostly straightforwardly implemented occurrences (such as JHotDraw 5.1 and 6.0, JUnit 3.7) as well as repositories that represent real big scientific applications and contain quite unusual pattern occurrences (such as PMD 1.8, Nutch 0.4). P-Mart stores results in the XML format.

Petterson et al. [54] summarized problems and challenges occurring during pattern detection tool evaluation and performed an extensive survey of pattern detection approaches. As an example the pattern detection tool CrocoPat http://www.sosy-lab.org/~dbeyer/CrocoPat/index.html was evaluated on the Java libraries SWT and Swing, detecting Singleton and Observer patterns. The results were saved in the benchmark DPDES. The main goal of this review was to make accuracy measurements more comparable. The survey of Pettersson clearly shows that the preconditions for a successful pattern detection benchmark - namely, a minimum level of maturity of the given research area and the existence of diverse approaches in the field - are met. A commonly agreed benchmark was suggested as a possible way to target all the above mentioned problems.

Dong et al. [17] compiled a thorough review of design pattern mining techniques. This review was built by analyzing of relevant articles, without running pattern detection tools. Different aspects of pattern detection approaches were compared: underlying program analysis (structural vs. behavioral, static vs.dynamic), supported programming languages (C++, Java, etc.), system representation (AST, matrix, etc.) and so on. The authors observed that different pattern detection approaches render different results. These observations were experimentally confirmed in Chapter 3 and served as the motivation for the data fusion approach (Chapter 4).

Note that all the benchmarks mentioned above (DEEBEE, P-Mart and DPDES) were formed by evaluating of pattern detection tools that use similar pattern detection techniques (for example, static structural analyses in DEEBEE) or using one tool at all (Ptidej in P-Mart and Crocopat in DPDES). This thesis contains an extensive evaluation of pattern detection tools that employ complementary pattern detection techniques (static vs. dynamic analysis; structural vs. behavioral analyses) and discusses the results with the authors of reviewed pattern detection tools and related articles.

10.5. Compiling Test Sets

Establishing a complete test set is a complicated, often unrealistic task. Ideally, to understand the intent of a particular occurrence, one needs to communicate directly to the author of this occurrence or to get a detailed documentation. These sources of information are usually unavailable. Pettersson et al. [53] and Dong et al. [17] observe that many repositories usually lack documentation. The situation is aggravated in big benchmarks that come with dependent libraries; finding all pattern occurrences is a too heavy task for a single human team.[1] Evaluating five different pattern detection tools experimentally, Kniesel et al. [17, 42] confirm experimentally the above mentioned observations.

Pattern detection experts typically develop *individual test sets* as follows:

- Pick straightforwardly implemented pattern occurrences with clear intent from well-discussed repositories (like JHotDraw `http://www.jhotdraw.org/`, discussions can be found in [18, 57]. This approach is taken, for example, in SSA [68] and PINOT [58].
- Use ad-hoc examples (for example, example code written by students or taken from book (Wang et al. [72]).

[1] I have found quite non-trivial occurrences even in widely known and well-discussed code repositories such as Java IO `http://oreilly.com/catalog/javawt/book/index.html` and JHotDraw `http://www.jhotdraw.org/`.

- Use occurrence sets from well-documented in-house projects (e.g. the authors of SPQR [36] collaborated with the authors of the Killer Widget library that was used for testing).

10.6. Data Fusion

To the best of the author's knowledge, the data fusion approach provided in Chapter 4 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 [56] 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 the approach from Chapter 4 that went one step further by combining the output of tools that already combine different techniques.

CHAPTER 11

Conclusions and Future Work

Main Conclusions. Behavioral aspects form the most crucial part of design pattern interactions and are more complicated than the structural ones. Joint efforts and discussions among several scholars were needed to collect a representative set of pattern occurrences and to capture their behavior using a generic specification framework (the Pattern Interaction Framework). This can be used as a basis for reasoning and comparing pattern occurrences and detection techniques.

The thesis presents a pattern detection tool DPJF that implements the concepts of the Pattern Interaction Framework. DPJF yields very high accuracy. False positives are discarded by the constraints that represent the behavioral characteristics. False negatives are largely avoided by relaxations of other constraints and, in particular, deeper exploration of transitive relationships.

DPJF demonstrated that a pattern detection algorithm based on a static behavioral analyses can be efficiently implemented while at the same time delivering very high accuracy. Practical observations show that scopes of behavioral analyses and of transitive relation depths are subject to empirically motivated limits. Together with a fine grained caching strategy, it makes the query time of DPJF faster than several of the existing ones, in spite of ideal precision and significantly enhanced recall. So DPJF has achieved a substantial advance in the state of the art of design pattern detection by improving precision and recall while keeping up with the fastest existing tools.

In such a way, DPJF removes several key obstacles that prevented professional developers from using pattern detection tools: bad precision, recall and speed. It is possible now to conduct empirical validations of the motivation behind pattern detection whereas at the same time investigating other factors that might also prevent adoption of pattern detection (e.g. belief that detection of patterns is of limited value, questioning of patterns in general).

Beyond the techniques for improving precision, recall and speed, the state of the art was also advanced by a methodological contribution: A more precise computation for precision and recall that assigns individual precisions and recall values to each reported candidate by counting its (reported and correct) role assignments.

Limitations of the current approach. At the theoretical level, only five collaborational patterns were specified using the framework provided in

Chapter 7. In addition, the framework only collaborational pattern specifications.

Some constraints should be refined taking into account the recent observations. For example, in the Control / Data Flow ordering constraint (Sec. 7.3) the data flow leading to the update of "Current State" field can be triggered not only within "setter methods".

At the practical level, DPJF supports only a few pattern specifications and a linuted number of implementation variants.

Ongoing work. The ongoing work is focused on interfacing DPJF to the public DPB design pattern benchmark repository [22]. To guarantee that the recall values computed by DPJF are based on validated test sets there is ongoing work on comparison the results obtained from DPJF's results to the results from DBP. In addition, the DPJF output is to be converted to the DBP format, so that other researchers can easily explore, verify and annotate DPJF results even without using DPJF.

Mid- and long-term continuations of the research outlined in the thesis include:

- Improving and developing DPJF:
  - Extending the set of supported design patterns and various implementation variants. In particular, creational patterns are to be specified.
  - Exploring further trade-offs between precision, recall and speed.
  - Experimenting with the underlying infrastructure - for example, the newest updates of JTransformer and SWI-Prolog.
  - Making DPJF collaborate with other software comprehension modules supported by JTransformer. For example, pattern candidates produced by DPJF can be fed into bad smell detectors which use the correlation between design patterns and bad smells in order to make more accurate judgements. This issue is being investigated in the ROOTS group `http://roots.iai.uni-bonn.de/research/`.
  - Supporting a user-friendly graphical interface that not only reports judgements but also advises a programmer how to transform false candidates into proper occurrences. JTransformer already includes the implementation of a permanently active development assistant that accepts judgments about wrong, incomplete or not recommended software artifacts and highlights them in a user-friendly graphical viewer. This *advisory* approach is a completely novel approach in software comprehension, pioneering the step from automated software comprehension to automated design improvement.
- Investigating the usage of auxiliary information (e.g. program element names, bad smells) witnessing the existence of design patterns. This is actual when detecting patterns whose solutions do not have

unique structural or behavioral characteristics (such as Adapter, Command, Builder, Flyweight) Additional information helps to understand the intention of a programmer thereby increasing the likelihood that a given candidate is a proper occurrence.

Bibliography

[1] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Design patterns: A round-trip. In *Proceedings of 11th ECOOP Workshop for PHD students in Object-Oriented Systems*, June 2001.

[2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: application to pattern detection and code synthesis. In *Proceedings of First ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[3] 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.

[4] Francesca Arcelli, Stefano Masiero, and Claudia Raibulet. Elemental design patterns recognition in java. *STEP'05*, 0:196–205, 2005.

[5] Francesca Arcelli, Marco Zanoni, and Andrea Caracciolo. A benchmark platform for design pattern detection. In *2nd International Conference on Pervasive Patterns and Applications (PATTERNS'10)*, 2010. Lisbon, Portugal, November 21-26.

[6] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *ESEC-FSE'07*, pages 385–394, New York, USA, 2007. ACM.

[7] Zsolt Balanyi and Rudolf Ferenc. Mining Design Patterns from C++ Source Code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM 2003)*, pages 305–314. IEEE Computer Society, September 2003.

[8] Ian Bayley and Hong Zhu. Formal specification of the variants and behavioural features of design patterns. *J. Syst. Softw.*, 83(2):209–221, 2010.

[9] Kent Beck and Ward Cunningham. Using Pattern Languages for Object-Oriented Programs. Technical report, September 1987.

[10] Kent Beck and Erich Gamma. *Contributing to Eclipse: Principles, Patterns, and Plug-Ins.* Addison-Wesley, Inc., Secaucus, NJ, USA, 1st edition, 2003.

[11] Alexander Binun and Günter Kniesel. Joining forces for higher precision and recall of design pattern detection. In *Proceedings of the 16th Conference on Software Maintenance and Reengineering (CSMR 2012), March 27-30*, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of java design patterns. In *ASE'01*, page 324, Washington, DC, USA, 2001. IEEE Computer Society.

[13] Iain Craig. *The Interpretation of Object-Oriented Programming Languages*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 2001.

[14] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice.* Addison Wesley, 2009.

[15] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. A two phase approach to design pattern recovery. In *CSMR '07*, pages 297–306, Washington, DC, USA, 2007. IEEE Computer Society.

[16] Jing Dong, Dushyant S. Lad, and Yajing Zhao. DP-Miner: Design pattern discovery using matrix. In *ECBS'07*, pages 371–380, Washington, USA, 2007. IEEE Computer Society.

[17] Jing Dong, Yajing Zhao, and Tu Peng. A review of design pattern mining techniques. *IJSEKE*, 2008.

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

[19] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society.

[20] Rudolf Ferenc, Arpad Beszedes, Lajos Fülöp, and János Lele. Design pattern mining enhanced by machine learning. In *ICSM'05*, pages 295–304, Washington, USA, 2005. IEEE Computer Society.

[21] Rudolf Ferenc, Juha Gustafsson, László Müller, and Jukka Paakki. Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa. *Acta Cybernetica*, 15:669–682, 2002.

[22] Francesca Arcelli Fontana, Andrea Caraciolo, and Marco Zanoni. DPB: A benchmark for design pattern detection tools. In *Proceedings of the 16th Conference on Software Maintenance and Reengineering (CSMR 2012), March 27-30*, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Francesca Arcelli Fontana, Stefano Maggioni, and Claudia Raibulet. Design patterns: a survey on their micro-structures. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[24] Lajos Jeno Fülöp. *Evaluating and Improving Reverse Engineering Tools*. Phd thesis, University of Szeged, Institute of Informatics, May 2011.

[25] Lajos Jeno Fülöp, Rudolf Ferenc, and Tibor Gyimothy. Towards a benchmark for evaluating design pattern miner tools. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 143–152, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Lajos Jenö Fülöp, Arpad Ilia, Adam Zoltan Vegh, and Rudolf Ferenc. Comparing and evaluating design pattern mining tools. In *Proceedings of SPLST '07*, 14th June 2007.

[27] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1994.

[28] G.Kniesel and A.Binun. Standing on the shoulders of giants - a data fusion approach to design pattern detection. In Andrian Marcus and Rainer Koschke, editors, *ICPC 2009*. IEEE, 2009.

[29] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *CASCON'04*, pages 28–41. IBM Press, 2004.

[30] Yann-Gaël Guéhéneuc, Herve Albin-Amiot, Remi Douence, and Pierre Cointe. Bridging the gap between modeling and programming languages. Technical report, Ecole des Mines de Nantes, 2002.

[31] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In *IJCAI'01*, pages 57–64, Seattle, USA, August 2001. AAAI Press.

[32] Yann-Gaël Guéhéneuc, Kim Mens, and Roel Wuyts. A comparative framework for design recovery tools. In *Proceedings of the Conference on Software Maintenance and Reengineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. *WCRE*, 0:172–181, 2004.

[34] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

[37] Olivier Kaczor, Yann-Gaël Guéhéneuc, and Sylvie Hamel. Efficient identification of design patterns with bit-vector algorithm. In *CSMR'06*, pages 175–184, Washington, USA, 2006. IEEE Computer Society.

[38] Holger Kampffmeyer and Steffen Zschaler. Finding the pattern you need: The design pattern intent ontology. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *MoDELS*, volume 4735 of *Lecture Notes in Computer Science*, pages 211–225. Springer, 2007.

[39] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[40] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. A role-based meta-modeling approach to specifying design patterns. In *COMPSAC '03: Proceedings of the 27th Annual International Conference on Computer Software and Applications*, page 452, Washington, DC, USA, 2003. IEEE Computer Society.

[41] Dae-Kyoo Kim and Lunjin Lu. Inference of design pattern instances in uml models via logic programming. In *ICECCS '06: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 47–56, Washington, DC, USA, 2006. IEEE Computer Society.

[42] Günter Kniesel and Alexander Binun. A data fusion approach to design pattern detection. Technical report IAI-TR-2009-02, ISSN 0944-8535, CS Department III, Uni.Bonn, Germany, January 2009.

[43] Günter Kniesel, Alexander Binun, Péter Hegedus, Lajos Jeno Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc, and Nikolaos Tsantalis. DPDX–towards a common result exchange format for design pattern detection tools. In Rafael Capilla, Rudolf Ferenc, and Juan C. Dueñas, editors, *CSMR*, pages 232–235. IEEE, 2010.

[44] Günter Kniesel, Alexander Binun, Péter Hegedus, Lajos Jenö Fülöp, Nikolaos Tsantalis, Alexander Chatzigeorgiou, and Yann-Gaël Guéhéneuc. A common exchange format for design pattern detection tools. Technical report IAI-TR-2009-03, ISSN 0944-8535, CS Department III, Uni.Bonn, Germany, October 2009.

[45] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *Proceedings of LATE '07*, New York, USA, 12th March 2007. ACM.

[46] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 114–134, London, UK, 1998. Springer-Verlag.

[47] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *Journal of Systems and Software*, 82(7):1177–1193, 2009.

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

[49] K. Meinke and J. V. Tucker, editors. *Many-sorted logic and its applications*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[50] Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Identification of behavioral and creational design patterns through dynamic analysis. *JSME*, 22(8), 2010. submitted.

[51] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

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

[53] Niklas Pettersson, Welf Löwe, and Joakim Nivre. On evaluation of accuracy in pattern detection. In *First International Workshop on Design Pattern Detection for Reverse Engineering (DPD4RE'06)*, October 2006.

[54] Niklas Pettersson, Welf Löwe, and Joakim Nivre. Evaluation of accuracy in design pattern occurrence detection. *IEEE Trans. Softw. Eng.*, 36:575–590, July 2010.

[55] Tomáš Poch and František Plášil. Extracting behavior specification of components in legacy applications. In *Proceedings of the 12th International Symposium on Component-Based Software Engineering*, CBSE '09, pages 87–103, Berlin, Heidelberg, 2009. Springer-Verlag.

[56] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE TSE*, 33(6):420–432, 2007.

[57] Dirk Riehle. Describing and composing patterns using role diagrams. In *Proceedings of the Ubilab Conference, 1996*, page 452, Zuerich, Switzerland, 1996. Universitaetsverlag Konstanz. Pages 137-152.

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

[59] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 74–83, Washington, DC, USA, 2003. IEEE Computer Society.

[60] J. Smith and D. Stotts. SPQR: Flexible automated design pattern extraction from source code. Technical report TR03-016, The University of North Carolina, CS Department, May 2003.

[61] Jason Smith and David 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, September 2003.

[62] Manu Sridharan. *Refinement-Based Program Analysis Tools*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2007.

[63] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. *SIGPLAN Not.*, 41(6):387–400, 2006.

[64] Krzysztof Stencel and Patrycja Wegrzynowicz. Detection of diverse design pattern variants. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 25–32, Washington, DC, USA, 2008. IEEE Computer Society.

[65] Bjarne Stroustrup. *The C++ Programming Language (Third Edition)*. Addison Wesley, 2004.

[66] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 281–293, New York, NY, USA, 2000. ACM.

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

[68] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE TSE*, 32(11):896–909, 2006.

[69] David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. Object, message, and performance: How they coexist in self. *Computer*, 25:53–64, 1992.

[70] Marek Vokác. An efficient tool for recovering design patterns from c++ code. *Journal of Object Technology*, 5(1):139–157, January 2006.

[71] M. von Detten and M. Platenius. Improving dynamic design pattern detection in reclipse with set objects. In *Proceedings of the 7th International Fujaba Days*, 2009.

[72] Wei Wang and Vassilios Tzerpos. Design pattern detection in eiffel systems. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 165–174, Washington, DC, USA, 2005. IEEE Computer Society.

[73] Patrycja Wegrzynowicz and Krzysztof Stencel. Towards a comprehensive test suite for detectors of design patterns. In *Proceedings of the ASE 2009, Auckland, New Zealand*, Nov 2009.

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

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

[76] Sven Wenzel and Udo Kelter. Model-driven design pattern detection using difference calculation. In *Proc. of the 1st International Workshop on Pattern Detection For Reverse Engineering (DPD4RE), co-located with 13th Working Conference on Reverse Engineering (WCRE'06)*, Benevento, Italy, October 2006.

APPENDIX A

Structurally similar patterns

In addition to the forward-to-parent group, the following similarity groups were observed:

- The group consisting of the Statically Typed Proxy pattern (Section 3.4). It is referred to as the *forward-to-sibling* group.
- The group consisting of the Singleton pattern. It is referred to as the *Singleton group* and is characterized by the following properties: a) the *Singleton* type owns a static field of the type Singleton; b) a static method owned by the *Singleton* type creates objects of the type *Singleton*
- The group consisting of the Prototype pattern. It is referred to as the *Prototype group* and is characterized by the following properties: a) a prototype class *Prototype* owns a field of the type *Prototype*; b) a method owned by the *Prototype* type creates objects of the type *Prototype*.
- The group consisting of Observer, Interpreter, Composite and Mediator patterns in which dependent objects (observers, children, colleagues) are kept in collection-typed variables. These patterns shared the following properties: a) callee methods are invoked on dependent objects in a loop and b) the field(s) pointing to the dependent objects can be updated in a non-construction method. Such a group is referred to as the *update-collection-field group*.
- The group consisting of State, Strategy, Bridge, Mediator and 1:1 Observer patterns. Dependent objects are kept in scalar-typed variables. These patterns shared the following properties: a) callee methods are invoked on dependent objects in a loop and b) the field pointing to the dependent objects can be updated in a non-construction method. Such a group is referred to as the *update-scalar-field group*.
- The group consisting of the Abstract Factory and Factory Method patterns referred to as *the creation group*. The following properties are shared: a) factory methods owned by some ”factory class” *FC* create objects of the type(s) different from *FC*.

The following list of similarity group EDPs was compiled:

- **Modify Static Field** (for the Singleton group) stands for updating of the static variable that point to the singleton object(s)

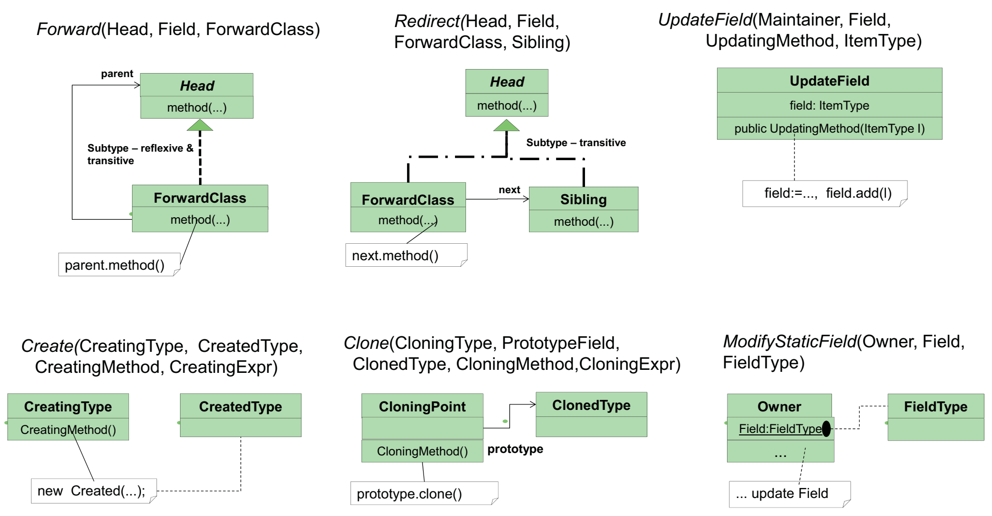

Forward(Head, Field, ForwardClass)
parent
Head
method(...)
Subtype – reflexive & transitive
ForwardClass
method(...)
parent.method()
Redirect(Head, Field, ForwardClass, Sibling)
Head
method(...)
Subtype – transitive
ForwardClass
method(...)
next
Sibling
method(...)
next.method()
UpdateField(Maintainer, Field, UpdatingMethod, ItemType)
UpdateField
field: ItemType
public UpdatingMethod(ItemType I)
field:=..., field.add(I)
Create(CreatingType, CreatedType, CreatingMethod, CreatingExpr)
CreatingType
CreatingMethod()
CreatedType
new Created(...);
Clone(CloningType, PrototypeField, ClonedType, CloningMethod,CloningExpr)
CloningPoint
CloningMethod()
prototype
ClonedType
prototype.clone()
ModifyStaticField(Owner, Field, FieldType)
Owner
Field:FieldType
...
FieldType
... update Field

FIGURE A.0.1. Similarity group EDPs

- **Clone** (for the Prototype group) stands for cloning of the "prototype" object.
- **Create** (in the creation group) stands for product object creations. In addition, it is actual in in the Proxy pattern, reflecting creations of real subject objects
- **Forward** (in the forward-to-parent group) stands for forwarding to the method belonging to the supertype
- **Redirect** (in the forward-to-sibling group) stands for forwarding to the method belonging to a sibling type
- **Update Field** is used in the update-scalar-field group and in the update-collection-field group. It denotes possible modifications of a field F within a public non-construction method. If F is scalar-typed, the variation of the Update Field referred to as the *Update Scalar Field* matches possible updates of the field pointing to the dependent objects in the State, Strategy, Bridge and 1:1 Observer patterns. If F is collection-typed, the *Update Collection Field* matches possible updates of the field pointing to the dependent objects.

The structure of each such EDP is presented in Figure A.0.1. Figure A.0.2. associates similarity group EDPs with the corresponding patterns.

Note that there design patterns mentioned in [27] that do not belong to similarity groups described in this thesis - for example, Adapter, Command, Builder, Visitor, Flyweight. In this case mining of EDPs is omitted when detecting these patterns.

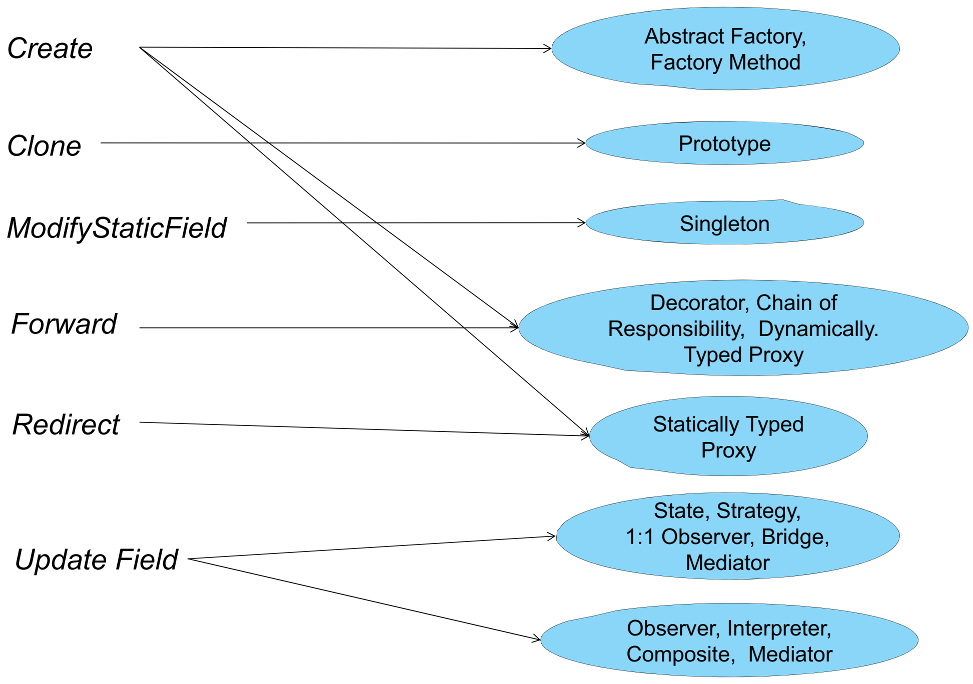

Create
Clone
ModifyStaticField
Forward
Redirect
Update Field
Abstract Factory, Factory Method
Prototype
Singleton
Decorator, Chain of Responsibility, Dynamically. Typed Proxy
Statically Typed Proxy
State, Strategy, 1:1 Observer, Bridge, Mediator
Observer, Interpreter, Composite, Mediator

FIGURE A.0.2. Using EDPs in pattern detection

APPENDIX B

Response Time Statistics

This section provides the relevant statistics (maximal and minimal value, up and low quartile and median) computed on the query times of Ptidej, SSA and DPJF. For these tools, query and intialization times can be computed separately (see Section 9.2). The statistics are computed on ten runs of each tool that are done on each benchmark from Sec. 3.1.

Figure B.0.1 (respectively, Figure B.0.2) contains six subtables that describe the query times of tools detecting the Decorator, the CoR and the Proxy patterns (respectively, the Observer and the Composite pattern). In each figure there is one subfigure for each benchmark. Each such subfigure contains the statistics for each tool that is capable of detecting the patterns from the corresponding similarity group. Columns stand for tools, rows stand for metrics (e.g. median, quartiles).

Note that in all cases DPJF yields the maximal query time on each benchmark and on each pattern *on the first run.* On subsequent runs the query time is smaller and corresponds to the median value.

Metric	Ptidej	SSA	DPJF
Maximum	5,99	0,21	*1,19*
Median	5,8965	0,2035	*1,067*
UpQuartile	5,94675	0,208	1,13575
LowQuartile	5,80475	0,197	1,032
Minimum	5,75	0,191	0,995

(A) JHotDraw 5.1

Metric	Ptidej	SSA	DPJF
Maximum	25,102	1,193	*5,5*
Median	24,8945	1,0525	*3,9295*
UpQuartile	24,916	1,0905	3,95525
LowQuartile	24,7128	1,02275	3,8735
Minimum	24,652	1,003	3,82

(B) JHotDraw 6.0

Metric	Ptidej	SSA	DPJF
Maximum	4,406	1,065	*1,15*
Median	4,081	1,009	*0,99*
UpQuartile	4,10525	1,05	0,994
LowQuartile	3,93925	0,988	0,97775
Minimum	3,438	0,972	0,97

(C) JavaIO

Metric	SSA	DPJF
Maximum	3,593	*9*
Median	3,552	*7,046*
UpQuartile	3,56025	7,06525
LowQuartile	3,50975	7,00625
Minimum	3,404	6,926

(D) AWT 1.14

Metric	SSA	DPJF
Maximum	5,194	*10,1*
Median	5,1355	*8,978*
UpQuartile	5,16625	9,0305
LowQuartile	5,1105	8,9505
Minimum	5,067	8,919

(E) ArgoUML 0.18.0

Metric	Ptidej	SSA	DPJF
Maximum	2,781	0,11	*0,65*
Median	2,705	0,1025	*0,5425*
UpQuartile	2,75	0,10625	0,54925
LowQuartile	2,66975	0,0935	0,539
Minimum	2,65	0,09	0,537

(F) Eclipse Team Core

FIGURE B.0.1. Response times for Decorator, Proxy and CoR

Metric	Ptidej	SSA	DPJF
Maximum	32,109	0,518	*0,72*
UpQuartile	31,9185	0,515	*0,527*
Median	31,8235	0,51	0,508
LowQuartile	31,6788	0,497	0,492
Minimum	31,641	0,481	0,485

(A) JHotDraw 5.1

Metric	Ptidej	SSA	DPJF
Maximum	70,113	3,089	*2,997*
UpQuartile	69,9943	3,031	*1,78*
Median	69,894	3,005	1,7465
LowQuartile	69,773	2,97675	1,68725
Minimum	69,652	2,92	1,651

(B) JHotDraw 6.0

Metric	Ptidej	SSA	DPJF
Maximum	2,438	1,087	*0,994*
UpQuartile	2,4115	1,02875	*0,255*
Median	2,345	1,006	0,2515
LowQuartile	2,2855	0,98125	0,24475
Minimum	2,157	0,912	0,242

(C) JavaIO

Metric	SSA	DPJF
Maximum	10,99	*8,813*
UpQuartile	10,0823	*7,1685*
Median	10,0385	7,1375
LowQuartile	9,93875	7,09175
Minimum	9,028	7,014

(D) AWT 1.14

Metric	SSA	DPJF
Maximum	205,143	*13,2*
UpQuartile	20,8655	*12,088*
Median	20,1235	12,0705
LowQuartile	20,0318	12,056
Minimum	19,824	12,046

(E) ArgoUML 0.18.0

Metric	Ptidej	SSA	DPJF
Maximum	7,359	0,112	*1*
UpQuartile	7,30925	0,10375	*0,4098*
Median	7,2795	0,1025	0,3995
LowQuartile	7,26525	0,09675	0,39525
Minimum	7,187	0,09	0,381

(F) Eclipse Team Core

Figure B.0.2. Response times for Observer and Composite