Université de Montréal

Identification of Behavioral and Creational Design Patterns through Dynamic Analysis

par
Janice Ka-Yee Ng

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Avril, 2008

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Identification of Behavioral and Creational Design Patterns through Dynamic Analysis

présenté par:

Janice Ka-Yee Ng

a été évalué par un jury composé des personnes suivantes:

Pierre Poulin
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Julie Vachon
membre du jury

Mémoire accepté le

RÉSUMÉ

Les patrons de conception proposent une solution à la fois simple et élégante aux problèmes récurrents en programmation orientée objet lors de l'implantation des programmes, car ils contribuent à améliorer la conception. Cependant, à l'usage, ces patrons sont disséminés dans le code source des programmes et, par conséquent, ne sont plus disponibles au moment de la maintenance. Pourtant, ils aideraient à comprendre leur implantation et conception, et à assurer la qualité des programmes après la maintenance.

Dans les travaux antérieurs, la structure et l'organisation des classes servent principalement de source de données pour l'identification des occurrences de patrons de conception. Il est toutefois intéressant de considérer la responsabilité des objets participant à l'exécution des programmes, puisque deux types de patrons de conception (comportementaux et créationnels) se caractérisent principalement par la distribution des responsabilités et la collaboration entre les objets à l'exécution.

Ce mémoire propose un méta-modèle et des algorithmes pour identifier automatiquement des occurrences de patrons comportementaux et créationnels dans le code source. Nous utilisons la méta-modélisation pour décrire les patrons de conception et les programmes Java, et l'analyse dynamique pour capturer le comportement des programmes au moment de l'exécution. La méta-modélisation permet d'expliciter la collaboration entre les participants impliqués dans l'exécution d'un programme (les messages) et de préciser leurs propriétés (condition d'exécution d'un message, répétition d'un message). Enfin, elle conduit à traduire les patrons en systèmes de contraintes avec explications et à identifier les occurrences similaires, formes complètes et approchées, par la résolution de problèmes de satisfaction de contraintes.

Mots clés : identification de patrons de conception, analyse dynamique, diagrammes de scénarios, programmation par contraintes, rétroconception.

ABSTRACT

The use of design patterns is a simple and elegant way to solve problems when designing object-oriented software systems because it leads to well-structured designs. However, after application, design patterns are lost in the source code, and are thus of little help during program comprehension and subsequent maintenance.

In previous work, the structure and organization among classes were the predominant source of data used for the identification of occurrences of design patterns. Yet, the responsibility of each participating object at runtime should not be neglected, as two types of design patterns, behavioral and creational, are mainly concerned with the assignment of responsibilities and the collaboration among objects at runtime.

This thesis proposes a metamodel and algorithms to automatically identify behavioral and creational design patterns in the source code. We use metamodelling to describe design patterns and software systems in Java, and dynamic analysis to capture the behavior of the systems at the moment of execution. The proposed metamodel allows the representation of interactions among the participants that take part in the execution of a system (messages) and their properties (conditions under which a message is executed, repetition of a message). Using this metamodel, the problem of behavioral and creational pattern identification can be translated into an explanation-based constraint satisfaction problem. Solving such kind of problems leads to exact or approximate occurrences of a design pattern.

Keywords: design pattern identification, dynamic analysis, scenario diagram, constraint programming, reverse engineering.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

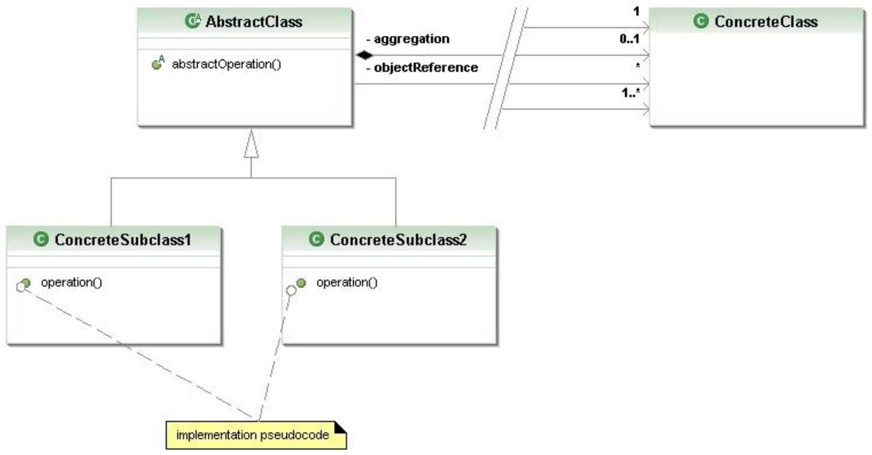
LIST OF ALGORITHMS

LIST OF APPENDICES

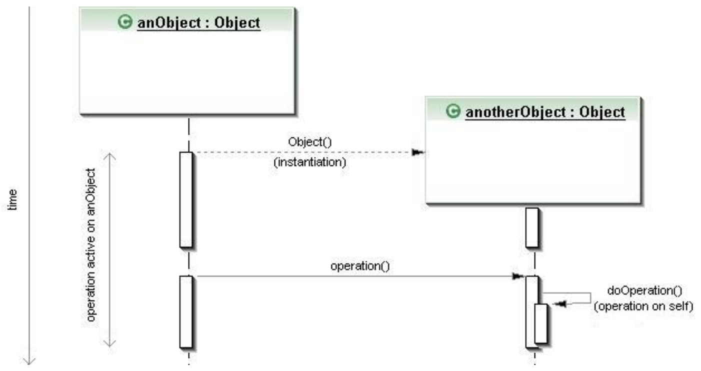
LIST OF ABBREVIATIONS

Bcel	Byte Code Engineering Library
CSP	Constraint Satisfaction Problem
eCSP	Constraint Satisfaction Problem with Explanations
Jikes RVM	Jikes Research Virtual Machine
OCL	Object Constraint Language
UML	Unified Modelling Language

NOTATION



Class diagram notation



Sequence diagram notation

送給爸爸、媽媽、哥哥、家姐和尚怡。

To Guillaume.

ACKNOWLEDGEMENTS

First, I would like to express my sincere gratitude to my greatly appreciated supervisor, Yann-Gaël Guéhéneuc, for giving me the chance to accomplish such a significant project, and supporting me unconditionally all along my studies. I would not have make it through without your sound advice ! Also, I would like to say thank you to each and every member of the GEODES team who contributed directly or indirectly to the success of my graduate studies. In particular, thank you the valuable professors of the team, Houari Sahraoui, Julie Vachon, and Petko Valtchev, whose words of wisdom are my source of inspiration. Moreover, thank you to Gerardo Cepeda, Simon Denier, Karim Dhambri, Sébastien Jeanmart, Foutse Khomh, Guillaume Langelier, Naouel Moha, Viet Thang Pham, Yousra Tagmouti, and Stephane Vaucher, with whom I had amusing discussions in the so-called 'salle-qu'il-ne-faut-pas-nommer' pantry, and above all, who made my life greedier than ever thanks to our regular visits to tasteful restaurants. Undeniably, I gained a few pounds with all those exquisite meals. However, I shared so many pleasant moments with the team !

Furthermore, I would like to express my deepest gratitude to Guillaume Roberge, who walked beside me since the beginning of my graduate studies, and believed in me under any circumstances. Your words of encouragement and perseverance inspired me to achieve passionately and assiduously my project. Without you, the road to graduation would have seem much more longer. Thank you for bearing with the unbearable me all these years !

最後，我不得不題一直關心我及支持我的家人。他們的存在是我在學校學習的動力。爸爸，感謝你這麼多年來的悉心栽培及經濟上的支持，要不然我跟本沒有能力完成這個課程。媽媽，感謝你晚晚煮又好味又有益的飯。哥哥，感謝你常常send短信來鼓勵我。家姐，感謝你每朝那麼早起身，都我是為了載我到地鐵站。尚怡，你是我肚子裡的蟲，你永遠都知道我在想什麼。沒有你們的打氣及精神支援，這篇文是不可能完成的。你們是最棒的！

CHAPTER 1

INTRODUCTION

1.1 Context: Reverse Engineering and Design Patterns

Software maintenance is a crucial phase of the software development process, as it consumes as much as 90% of the total resources dedicated to a software system [Erl00]. The main activity during maintenance is program comprehension, during which maintainers analyze the structure and organization of code artifacts with the help of re-engineering tools, to perform maintenance tasks such as debugging or adding new features. The recovery of the design and architecture is relevant to the maintainers, as they provide insights on the original choices made during the initial phase of conception.

Since their inception in 1994, design patterns [GHJV94] have been increasingly used to design and obtain well-structured systems. Design patterns are solutions to recurrent problems in object-oriented systems. Their recovery during the maintenance should consequently make the task easier. However, in reality, design patterns are lost in the source code of the systems due to their complexity and lack of documentation. It is difficult or impossible to manually recover design patterns applied during system design and implementation, which impedes program comprehension and increases the cost of its maintenance [ACdPF01].

1.2 Motivations: Identification of Behavioral and Creational Patterns

While organizing the catalog of design patterns, Gamma *et al.* [GHJV94] divided their patterns into three families of related patterns. Following the criterion of *purpose*, which indicates what the pattern does, patterns can either have a **structural**, **behavioral**, or **creational** purpose (*cf.* Annexe I for the classification). Structural patterns deal with the organization/composition of classes or objects. Behavioral patterns characterize the ways in which objects interact and share responsibility. Creational patterns are concerned with object creation.

Several approaches have been proposed to identify design patterns in source code using static analysis, for example [KP96,SvG98,Wuy98,KSRP99,Nie02,PL06, TCSH06]. The fundamental idea of these approaches consists in analyzing the class structure of a system to identify micro-architectures that are similar to the known structure of a design pattern. These approaches are limited when recovering pure behavioral and creational patterns because the responsibilities and the collaboration among objects at runtime cannot be easily and completely determined using static analysis. Moreover, behavioral and creational design patterns can hardly be described only by their structure.

Thus, to provide a better view of the system to be reverse engineered, previous work suggested to combine static and dynamic analyses, because the dynamic aspect can provide data to complement those related to the structure of a system (as shown in [HHL02,HLM03]). First, static analysis is performed to find pattern candidates that have the structure of the searched pattern; then, dynamic analysis is conducted to remove false positives, and thus to confirm pattern candidates. However, the limitation still remains: behavioral and creational patterns, which cannot all be uniquely identified by structural means, risk not to be recovered during static analysis.

Therefore, in this thesis, we propose a pure dynamic approach that directly analyzes the collaboration among objects by making use of dynamic analysis for the recovery of behavioral and creational patterns.

1.3 Contributions

Although the domain of design pattern identification is well-established, we contribute to this domain with an approach that identifies behavioral and creational design patterns. Using existing sequence diagram recovery and explanation-based constraint programming techniques, our approach can identify pure behavioral and

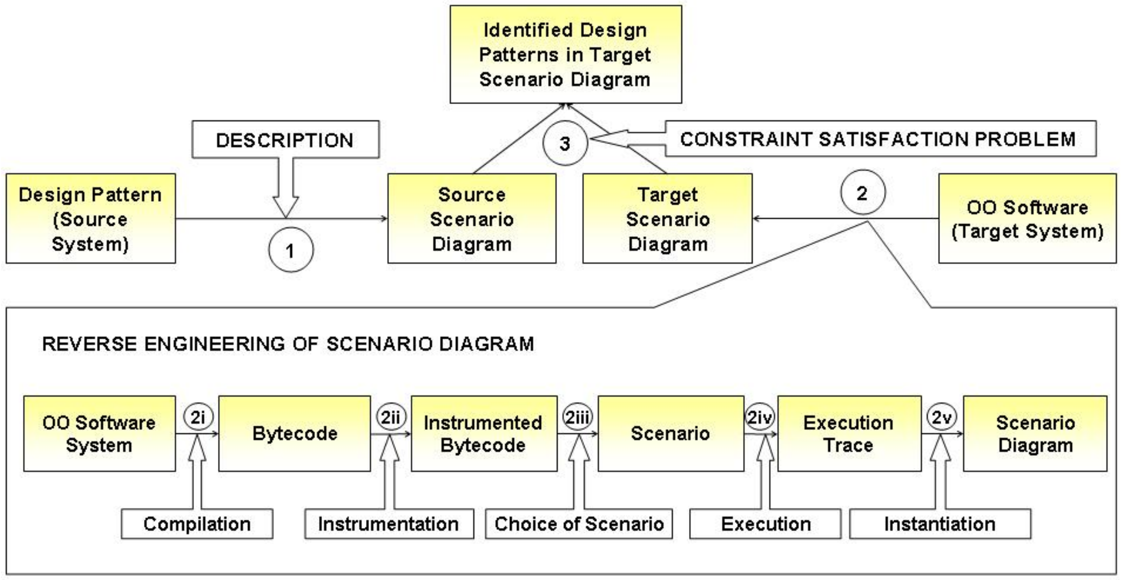


Figure 1.1: A 3-step approach for the identification of design patterns through dynamic analysis

creational patterns, unlike other previous work which focused on structural patterns recovery.

We propose a 3-step approach (as illustrated in Figure 1.1, Steps 1, 2, and 3) to identify behavioral and creational design patterns in source code using dynamic analysis. First, we describe behavioral and creational patterns in terms of UML-like sequence diagrams. In our approach, UML sequence diagrams are obtained from the execution of a use case. Therefore, these diagrams will be referred to as *scenario diagrams* in the rest of this thesis: they are only partial UML sequence diagrams describing one scenario corresponding to a use case, instead of all possible alternatives for the exercised use case [BLL06].

Second, using dynamic analysis, we reverse engineer a dynamic model—again, in the form of scenario diagrams—of the collaboration among objects of a system for a given scenario. In this context, we are interested in discovering the exact collaboration among runtime objects to find *real* occurrences of a design pattern (as opposed to *potential* ones). Therefore, we favored dynamic analysis over static analysis to obtain dynamic data. Previous work (even the currently most powerful ones [RC05]) showed that interaction diagrams, such as UML collaboration or scenario diagrams generated from static analysis, result in possibly highly inaccurate diagrams and, in the worst case, describe impossible behaviors, depending on the technique used to determine object references in the source code [TP03, RC05, RVR05]. Furthermore, it is not conceivable for any large, complex systems to perform coarse-grained and sophisticated analysis of the source code to determine the dynamic types of object references [GJM91]. In contrast, dynamic analysis is more accurate because it reports precisely the interactions between objects without symbolic representation. By tracing the execution of a use case, we easily obtain data that can be used to reverse engineer a scenario diagram. However, its main limitation is that it depends on the executed scenario. We plan to address, in future work, the building of a complete sequence diagram for a given use case from a set of scenario diagrams. This building requires triggering as many scenarios as possible through multiple executions of a system, and their analysis to merge them into one sequence diagram.

Finally, as in previous work [GAA01], we translate the problem of design pattern identification into a constraint satisfaction problem with explanations (eCSP), which consists in assigning concrete objects and messages from the scenario diagram of the executed scenario, to the roles in the scenario diagram of a design pattern. Solving the eCSP consists in matching, one against the other, the objects and messages of the scenario diagram of a design pattern with the ones of the executed scenario. We chose to use explanation-based constraint programming to solve the problem of pattern identification because it allows both the identification of complete and approximate occurrences of design patterns, and allows interaction guidance.

We can resume the contributions of this thesis in the following points:

- An approach to automatically identify pure behavioral and creational design patterns through dynamic analysis;

- A technique based on metamodelling and intermediate code instrumentation to reverse engineer the scenario diagrams of an object-oriented system;
- The building of a library of specialized constraints to describe the collaboration among objects in terms of concepts introduced by our scenario diagram metamodel.

Last but not least, we had the opportunity to publish the results of our research in [NG07].

Hypothesis
Using dynamic analysis, is it possible to identify behavioral and creational design patterns in object-oriented software systems?

1.4 Outline

This thesis is structured as follows: Chapter 2 presents the related work. Chapter 3 introduces our metamodel to capture the collaboration among objects at runtime. This metamodel is similar to the UML metamodel for scenario diagrams. It is used to describe behavioral and creational design patterns, as well as scenario diagrams from any scenario of a system. Chapter 4 describes our technique to reverse engineer sequence diagrams of an object-oriented system. Chapter 5 presents the technique used to identify behavioral and creational design patterns. Chapter 6 illustrates our identification approach using one scenario of JHotDraw, a drawing editor for technical and structured graphics, then reports results related to the identification of the Builder, Command, and Visitor patterns in five systems, and finally discusses the approach. We conclude and present future work in Chapter 7.

CHAPTER 2

RELATED WORK

The identification of design patterns in object-oriented systems has been the subject of many work. In particular, the identification of structural design patterns has been investigated as early as 1998 [Wuy98]. However, we are not aware of work dedicated to the identification of behavioral and creational patterns (without any structural data). Thus, we present work related to the identification of structural design patterns in Section 2.1, and to the use of dynamic data during structural design pattern identification in Section 2.2. These identification approaches are compared one against the other according to the following criteria:

- The types of recovered design patterns (structural, behavioral, and creational);
- The possibility to get explanations on the obtained occurrences;
- The degree of automation of the identification technique;
- The accuracy of the precision and recall measures of the identification technique;
- The performance of the identification technique;
- The ability to deal with variants/approximations of the design patterns.

The necessity for identification approaches to deal with variants/approximations of patterns originates from the fact that the implementation of a design pattern in most systems, although true to their original intention in [GHJV94], does not always strictly match the solution as described in theory. In this chapter, we explore the different techniques of pattern identification in this respect, among others. Finally, we dedicate Section 2.3 to work related to the recovery of sequence diagrams, providing guidance for our own reverse engineering approach of scenario diagrams.

2.1 Structural Pattern Identification

Since their inception in 1994, design patterns have been the subject of many work related to their recovery in software systems. We present here four characteristic contributions that all use static data, but distinguish themselves by their identification technique: Prolog, queries, eCSP, and matrices similarity.

For example, Wuyts [Wuy98] published a precursor work on structural design pattern identification. His approach consisted in representing object-oriented systems as Prolog facts and predicates: facts describe the structure of object-oriented systems, such as inheritance and acquaintance relations, while predicates are used to formulate queries for reasoning about facts and to identify occurrences of design patterns.

Facts were extracted using static analysis. The Composite pattern is expressed manually using the predicates depicted in Figure 2.1. Briefly, a composite pattern consists in the definition of a certain structural relationship between the variables of component `?comp` and composite `?composite`. Also, there is an aggregation relationship between these two. The `compositeStructure` rule defines that `?comp` is a class, and that `?composite` is a subclass, direct or indirect, from the composite. The `compositeAggregation` predicate expresses that the composite should override at least one method of the component, and in this overridden method, it should do an enumeration over the instance variable that holds these composites and recursively apply the method `?msg` to each of the composites. This approach had performance limitations due to the use of a Prolog engine. It could not deal with variants automatically, and showed limited precision and recall according to subsequent studies.

This previous work was followed by many other approaches to improve on its performances. These following approaches use different data and different representation and detection techniques. For example, Quilici *et al.* [QYW97] established a relationship between plan recognition and program comprehension. Plan recognition makes use of structural events and actions to determine "*the best unified* *context which causally explains a set of perceived events as they are observed*". The authors derived a new approach to program plan recognition by augmenting existing AI plan recognition algorithms. In particular, their experimentations check the presence of instances of a given plan in the system source code using constraint programming, while improving the performance of plan recognition. However, poor performances are again an issue: results show that constraint checks, *i.e.*, their measure of efficiency, deteriorates with the number of lines of code.

Guéhéneuc *et al.* [GJ01] drew inspiration from previous work to propose an approach for structural design pattern identification. This problem is represented as a constraint satisfaction problem, for which the authors introduced the use of explanation-based constraint programming, and a dedicated metamodel capturing structural representation of both design patterns and systems [AAG01] as depicted in Figure 2.2. It includes the modelling of binary class relationships such as associations, aggregations, and compositions [GAA04] to improve both the representation and the handling of variants.

Recently, Tsantalis *et al.* [TCSH06] introduced a measure of similarity between matrices representing either systems or patterns to improve performance and deal with a wide range of structural design patterns, as well as their modified versions.

```
RULE
    head: compositePattern(?comp,?composite,?msg)
    body: compositeStructure(?comp,?composite)
          compositeAggregation(?comp,?composite,?msg)
RULE
    head: compositeStructure(?comp,?composite)
    body: class(?comp)
          hierarchy(?comp,?composite)
RULE
    head: compositeAggregation(?comp,?composite,?msg)
    body: commonMethods(?comp,?composite,?M,?compositeM)
          methodSelector(?compositeM,?msg)
          oneToManyStatement(?compositeM,?instVar,?enumStatement)
          containsSend(?enumStatement,?msg)
```

Figure 2.1: The Composite pattern described as Prolog facts and rules by Wuyts [Wuy98]

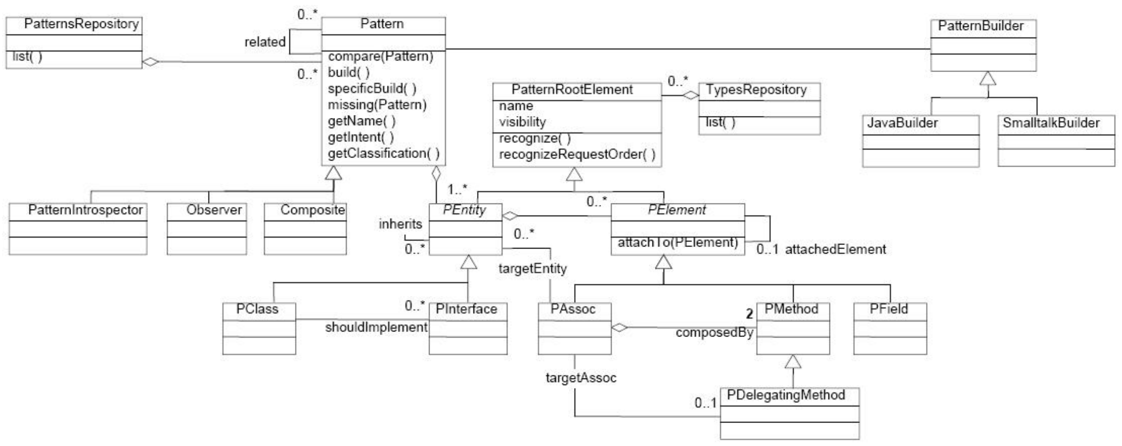


Figure 2.2: A design pattern metamodel proposed by Guéhéneuc *et al.* [GJ01]

The structure of systems or patterns is modelled using matrices, because the authors consider that a class diagram is fundamentally "*a directed graph that can be perfectly mapped into a square matrix*". Each kind of information such as associations, generalizations, abstract classes, object creations, or abstract method invocations is represented in an individual graph/matrix. Figure 2.3 shows such a representation for the Decorator pattern.

2.2 Dynamic Data for the Identification of Patterns

Some work suggested to improve the precision and recall of previous approaches by combining static and dynamic identifications.

For instance, Heuzeroth *et al.* [HLM03] proposed an approach that uses both static and dynamic data to identify interaction patterns. On the one hand, the static specification of a pattern is a collection of Prolog predicates that describe the relations between the elements, as illustrated in Figure 2.4. On the other hand, the dynamic aspect is also represented by Prolog predicates, following the temporal logic of actions. Static analysis of the system source code is performed under the form of Prolog query corresponding to the static predicates, to suggest a set

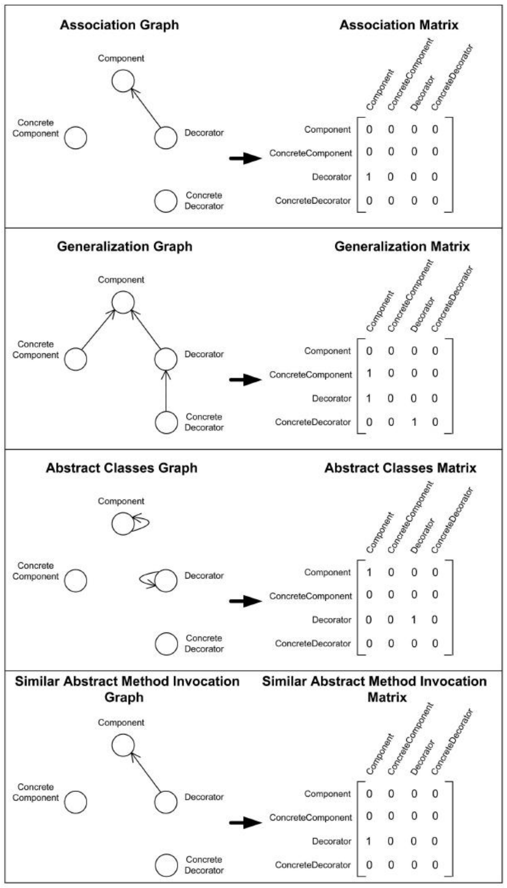


Figure 2.3: Structural representation of the Decorator pattern by Tsantalis *et al.* [TCSH06]

of candidate pattern instances conforming to the static structure of the searched pattern. Then, dynamic analysis monitors the execution of the candidates, and validates their behavior according to the dynamic specifications of the searched pattern, as shown in Figure 2.5. Finally, a dedicated validator confirms or rejects the candidates. The proposed approach was exemplified with the Observer, Composite, and Decorator patterns, and evaluated on their own analysis tool system. However, the authors did not provide details about the measures of precision and recall. It is therefore difficult to determine the impact of false negatives omitted during the static analysis on the results of the experimentations, and possible al-

```
observer(Vattach, Vattachee, Vdetach, Vdeteachee, Vlistener,
         Vnotify, Vsubject, Vupdate):-
          listener(Vlistener, Vupdate),
          subject(Vattach, Vattachee, Vdetatch, Vdetachee, Vlistener,
                  Vnotify, Vsubject, Vupdate).

subject(Vattach, Vattachee, Vdetach, Vdetachee, Vlistener,
        Vnotify, Vsubject, Vupdate):-
          notify(Vnotify, Vsujbect, Vupdate),
          attach(Vattach, Vattachee, Vlistener, Vsubject),
          detach(Vdetach, Vdetachee, Vlistener, Vsubject),
          class(Vsjubect).

attach(Vattach, Vattachee, Vlistener, Vsujbect):-
       attachee(Vattachee, Vlistener),
          assignAttachee(Vattachee, Vstatement15),
          member(Vattach, Vsubject),
          method(Vattach),
          parameter(Vattachee, Vattach),
          statement(Vstatement15, Vattach
```

Figure 2.4: The Observer pattern described statically by Heuzeroth *et al.* [HLM03]

```
watch('attach', Vattach, Arguments):-
   observer(Vattach, Vattachee, _, _, _, _, _, _),
   Arguments = ['this', Vattachee].

onMethodEntry('attach', Vattach, [Vsubject, Vattachee]):-
   dynamicConformTyped(Vattachee, VattacheeClass),
   containingTyped(VattacheeName, VattacheeClass),
   dynamicConformTyped(Vsubject, VsubjectClass),
   dynamicObserver(VNo, Vattach, VattacheeName, _, _, _, _, VsubjectClass, _),
   request(assert(attached(VNo, Vsubject, Vattachee))),
   fail.
```

Figure 2.5: Validation of the behavior of methods by Heuzeroth *et al.* [HLM03]

ternatives to deal with. Also, it is unclear how this approach can be generalized to pure behavioral and creational design patterns. Finally, as the proposed approach uses a Prolog engine for querying facts, it had the same performance limitations as Wuyts.

Shawky *et al.* [SAEHES05] proposed a similar approach to improve the precision and recall of a static identification approach. During the static analysis that determines the pattern candidates fulfilling the static characteristics of a pattern, the authors introduced two constraints, *i.e.*, a method delegation constraint and a key method constraint, to reduce the number of pattern candidates that the dynamic analysis must validate. On the one hand, the method delegation constraint statically checks for specific method invocations. On the other hand, the key method constraint checks whether the objective and behavior of a pattern at runtime are fulfilled. Only pattern candidates satisfying the aforementioned static constraints are provided as inputs to the dynamic analysis. However, it is unclear which dynamic constraints a pattern candidate should satisfy to be validated. Also, this approach uses a customized debugger to manually inspect the dynamic behavior of a key method, which decreases considerably the degree of automation of its dynamic analysis.

Most of the previous work on the identification of structural design patterns uses data related to method calls, which can be considered as dynamic data. For example, Antoniol *et al.* [ACdPF01] used delegation constraints to further reduce the set of pattern candidates satisfying the structural specification of a design pattern. Guéhéneuc *et al.* [GJ01], as mentioned in the previous section, recovered, among other binary relationships, composition relationships between classes [GAA04] by tracing three specific events when a system is executed: assignment events, finalize events, and system-end events.

2.3 Recovery of Sequence Diagrams

The recovery of scenario diagrams has been tackled by several authors. An important contribution to this domain is the work of De Pauw *et al.* [PKV94], which describes a model to visualize data about the execution of object-oriented systems. In representing the dynamic behavior of the systems, the authors have chosen to use a canonical four-dimensional event space to conceptually model object construction and destruction, and method invocation and return, as illustrated in Figure 2.6. Each point in the space is described by the coordinate quadruple (*class*, *instance*, *method*, *time*), and corresponds to an event during system execution.

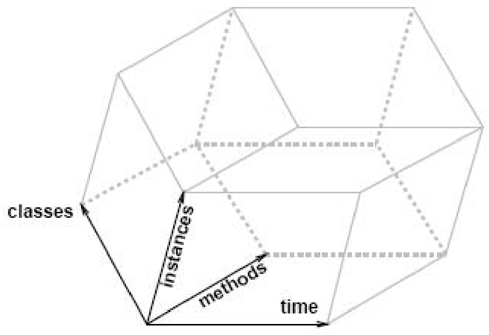


Figure 2.6: Canonical four-dimensional event space by De Pauw *et al.* [PKV94]

However, the implementation of such a model is impractical, as a typical system can generate many hundreds of thousands of construction, destruction, enter, and leave events. To overcome this limitation, the authors made a correlation between this event space and the notion of *call frames* to store combinations of events, to ensure that execution data is stored compactly and information retrieval is efficient. From there on, users can navigate and visualize the event space respectively using queries and the proposed visualizing tool, allowing the exploration of the different perspectives, as illustrated in Figures 2.7, 2.8, and 2.9. On the one hand, the inter-class call cluster provides a dynamic overview of communication patterns between classes. On the other hand, the inter-class call matrix gives cumulative and more quantitative information. Finally, the histogram of instances displays all instances of each class. Later, in 2002, De Pauw *et al.* [PJM$^+$02] designed a tool to visualize many facets of system behavior. In particular, a technique for pattern extraction is proposed to simplify the views by eliminating repetitions of messages. However, it is not clear whether a detected repetition of messages distinguishes the execution of a loop from the incidental execution of identical method sequences in different contexts.

Rountev *et al.* [RVR05] described a first algorithm to reverse engineer UML 2.0 sequence diagrams by control-flow analysis of Java code. One of their objectives was to represent the intraprocedural behavior of the systems, such as conditional and

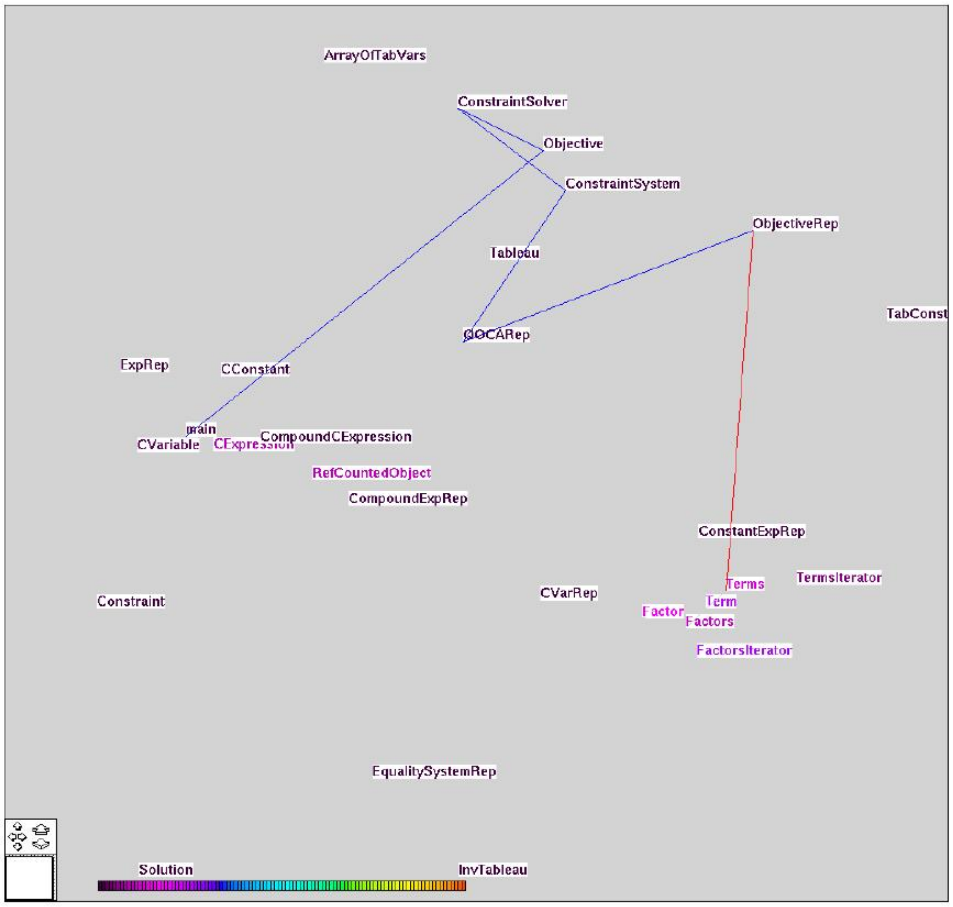


Figure 2.7: Inter-class call cluster by De Pauw *et al.* [PKV94]

iterative behaviors, in the reverse engineered sequence diagram. As a result, they extended UML 2.0 by generalizing the *break fragment* to allow breaking out multiple enclosing fragments and by defining the *return fragment* to model multiple exit nodes. For each method of a system, the approach transforms its control flow graph into a partial sequence diagram. Then, a single multi-method diagram is generated by combining these partial diagrams. Figure 2.10 shows such a transformation. Their approach does not consider data obtained by dynamic analysis, and thus solely depends on the accuracy of the control-flow analysis.

Briand *et al.* [BLM03] proposed a method to reverse engineer scenario diagrams from execution traces. In particular, they proposed two metamodels, one describing

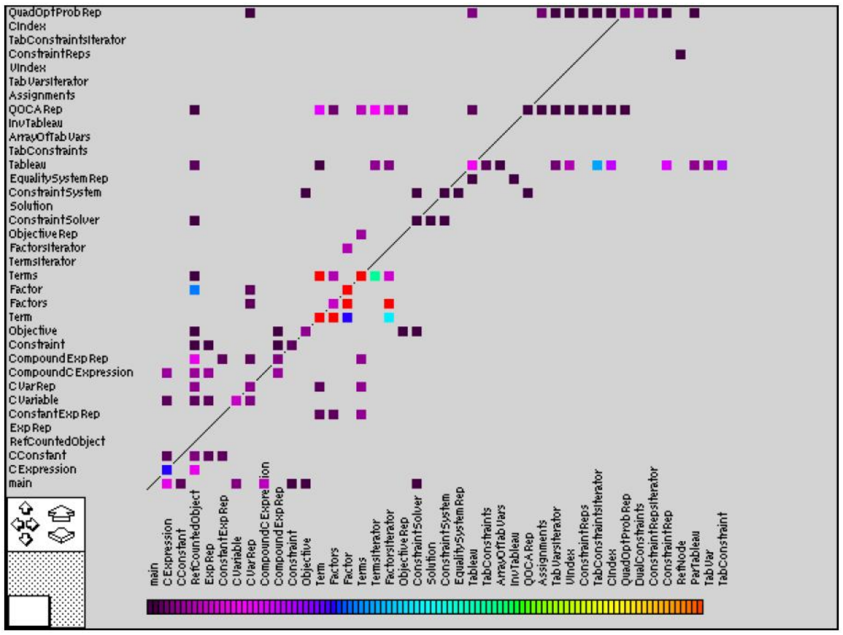


Figure 2.8: Inter-class call matrix by De Pauw *et al.* [PKV94]

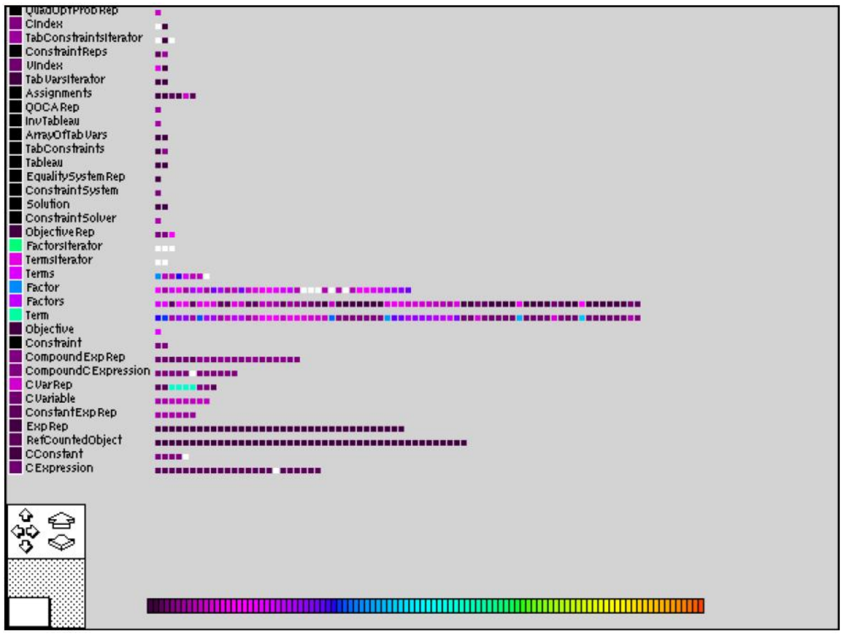


Figure 2.9: Histogram of instances by De Pauw *et al.* [PKV94]

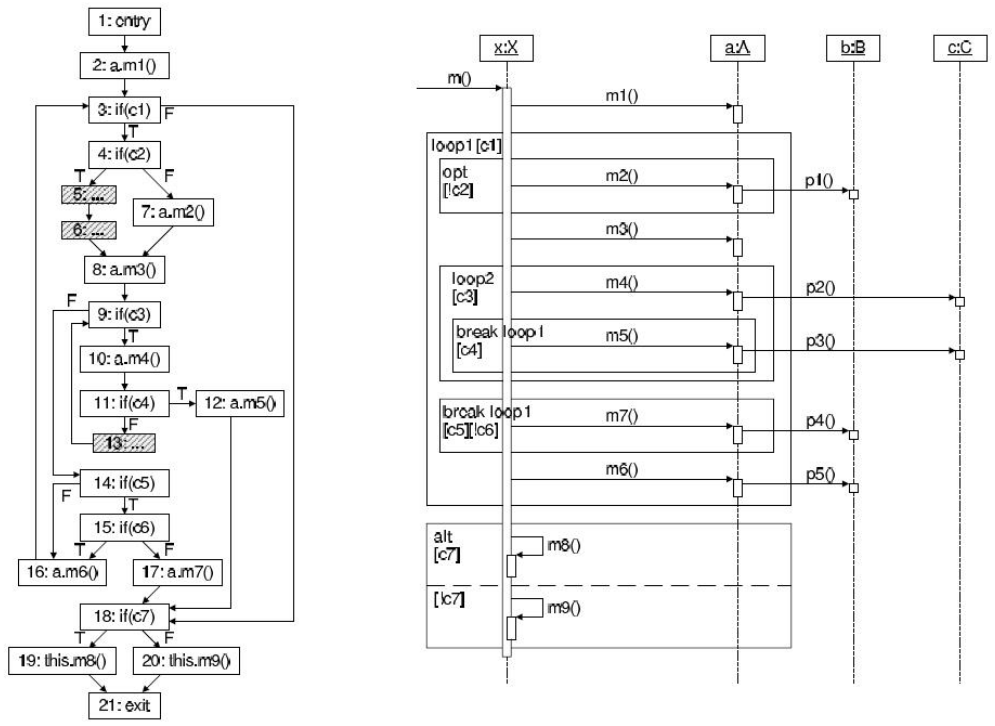


Figure 2.10: CFG for a method m and its corresponding reverse engineered scenario diagram by Routnev *et al.* [RVR05]

scenario diagrams, and the other describing execution traces. A mapping established between the two metamodels, illustrated by Figures 2.11 and 2.12, defines how to derive a scenario diagram from a recovered execution trace. This mapping is defined by a set of consistency rules in OCL, like the ones shown on Figure 2.13. Later, in 2006, Briand *et al.* [BLL06] introduced a complete approach to recover scenario diagrams using execution traces. Their work inspired our own recovery approach.

2.4 Conclusion

In this chapter, we covered related work of three important fields that inspired our behavioral and creational pattern identification approach. First, we presented work on structural pattern identification. Second, we reviewed approaches that

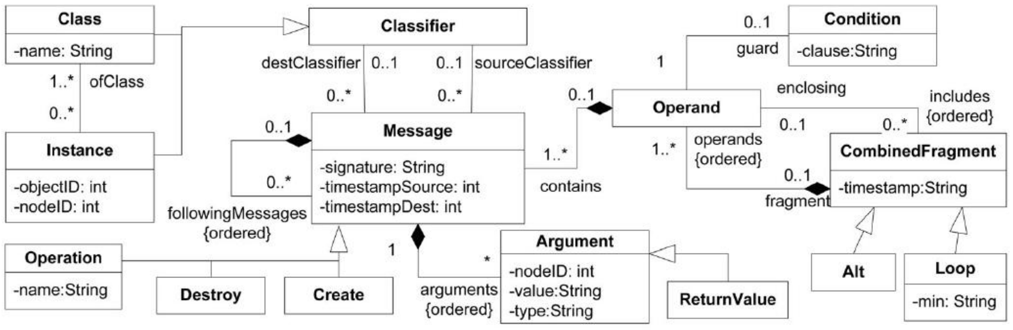


Figure 2.11: Scenario diagram metamodel by Briand *et al.* [BLM03]

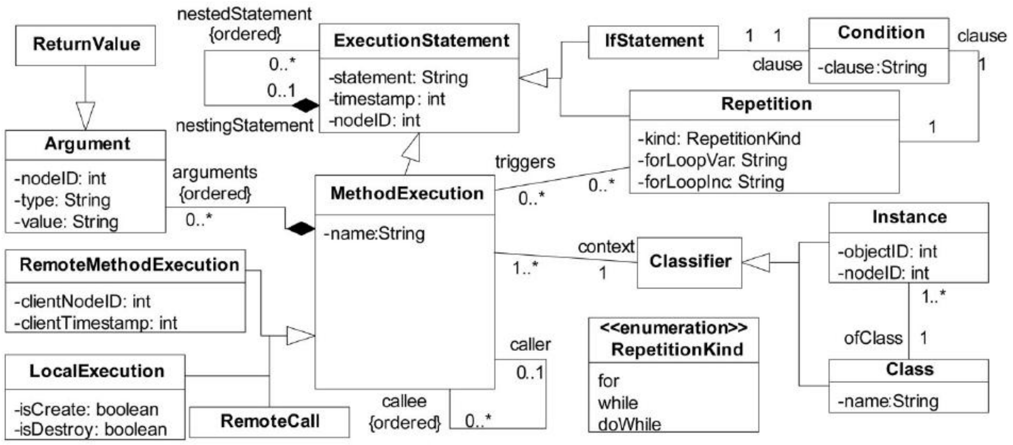


Figure 2.12: Execution trace metamodel by Briand *et al.* [BLM03]

```
methodCall.allInstances->forAll(m1,m2: MethodCall | m1.callee->includes(m2)
implies
  methodMessage.allInstances->exists(mm:MethodMessage |
    mm.content = m2.statement
    and
    if m1.context.oclType = InstanceTRACE then (  // checking the caller
      mm.callerObject.addressID =m1.context.id and
      mm.callerObject.theClass.name = m1.context.theClass.name
    ) else (
      mm.callerObject.name = m1.context.name
    )
    and
    if m2.context.oclType = InstanceTRACE then (  // checking the callee
      mm.calleeObject.addressID = m2.context.id and
      mm.calleeObject.theClass.name = m2.context.theClass.name
    ) else (
      mm.calleeObject.name = m2.context.name
    )
    and
    mm.returnType = m2.returnType and
    mm.parameterSEQD->forAll(index:Integer |      // checking the parameters
      mm.parameterSEQD->at(index).name = m2.parameterTRACE->at(index).name and
      mm.parameterSEQD->at(index).type = m2.parameterTRACE->at(index).type
    ) and mm.parameterSEQD->size = m2.parameterTRACE->size
    and      // checking the conditions
    if m2.precedentStatement.oclType = ConditionStatement then (
      mm.clause->forAll(index:Integer |
        mm.clause->at(index).clauseStatement =
          m2.precedentStatement.clauses->at(index).clauseStatement and
        mm.clause->at(index).clauseKind =
          m2.precedentStatement.clauses->at(index).clauseKind
      ) and mm.clause->size = m2.precedentStatement.clauses->size
    and
    // checking repeated executions
    if ( m2.precedentStatement.oclType = ConditionStatement and
        m2.precedentStatement.isLoop = true and
        m2.precedentStatement.followingMessage->size =1 ) then
      mm.timesOfRepeat = methodCall.allInstances->select(m3:MethodCall|
        m3.statement = m2.statement and m3.returnType = m2.returnType and
        m3.caller = m2.caller and
        m3.parameterTRACE->forAll(index:Integer|
          m3.parameterTRACE->at(index).name = m2.parameterTRACE->at(index).name and
          m3.parameterTRACE->at(index).type = m2.parameterTRACE->at(index).type
        ) and
        m3.parameterTRACE->size = m2.parameterTRACE->size and
        m3.precedentStatement.oclType = ConditionStatement and
        m3.precedentStatement.clauses->forAll(index:integer|
          m3.precedentStatement.clauses->at(index).clauseStatement =
            m2.precedentStatement.clauses->at(index).clauseStatemenr) and
          m3.precedentStatement.clauses->at(index).clauseKind =
             m2.precedentStatement.clauses->at(index).clauseKind)
         and m3.precedentStatement.clauses->size = m2.precedentStatement.clauses->size
      )->size
    and
    mm.followingMessage->forAll(mm1:Message| mm1.callerObject = mm.calleeObject)
     )
)
```

Figure 2.13: Consistency rules for the mapping between two metamodels by Briand *et al.* [BLM03]

combined static and dynamic data. Finally, we presented different techniques for the recovery of sequence or scenario diagrams from execution traces.

This thesis do not pretend to provide a new approach for the recovery of structural patterns in legacy systems, which is already a well-established domain of research. Instead, we gather existing techniques for sequence diagram and pattern recovery, and propose a dynamic approach that is complementary to design pattern identification approaches based on static analysis. Our dynamic approach focuses primarily on pure behavioral and creational patterns that cannot be recovered by most of the previous approaches because dynamic data is mandatory for their identification. Furthermore, if a static analysis must be performed afterwards to reduce the number of pattern candidates suggested by our dynamic analysis, a much less heavy structural analysis is required since the only things left to verify are the inheritance and binary class relationships among classes, as opposed to previous approaches on the identification of structural patterns.

In the next chapter, we will tackle the problem of modelling interactions between objects in an object-oriented software, by providing the description of a metamodel that we will use to model object responsibilities and interactions. This metamodel is fundamental for the modelling of the objects' collaboration suggested by behavioral and creational patterns, as well as for the modelling of the various system scenarios.

CHAPTER 3

SCENARIO DIAGRAM AND DESIGN PATTERN DESCRIPTION

In this chapter, we first present the conventional UML notation adopted in literature to describe scenario diagrams (Section 3.1). We also introduce the scenario diagram metamodel (based on the UML metamodel), which we use to express the dynamic aspects of the design solutions advocated by behavioral and creational design patterns. Then, Section 3.2 provides the description of behavioral and creational patterns as suggested by [GHJV94].

3.1 Scenario Diagram Notation and Metamodel

UML 2.0 proposes the use of interaction diagrams to describe how the objects of a system handle operations and behaviors. Usually, one comprehensive interaction diagram is built for each use case to describe the sequences of messages and operations that realize the system overall functionality. A scenario diagram is an interaction diagram that describes how messages flow from one object to another. They show the order in which requests between objects get executed by means of objects, lifelines, and messages among objects—respectively corresponding to boxes, vertical lines, and arrows. This type of diagram may also contain additional information about the flow of control during the collaboration, such as *if-then* conditions ("*if* `c` *then* send message `m`") and loops ("send message `m` multiple times").

Time flows from top to bottom in a scenario diagram. A solid vertical line indicates the lifetime of a particular object. The objects are named *a : aSomething*, where *a* is the object name, and *aSomething* is the class of the object. If an object is dynamically created during a scenario then its lifeline starts at the vertical point indicating the time at which the object was created.

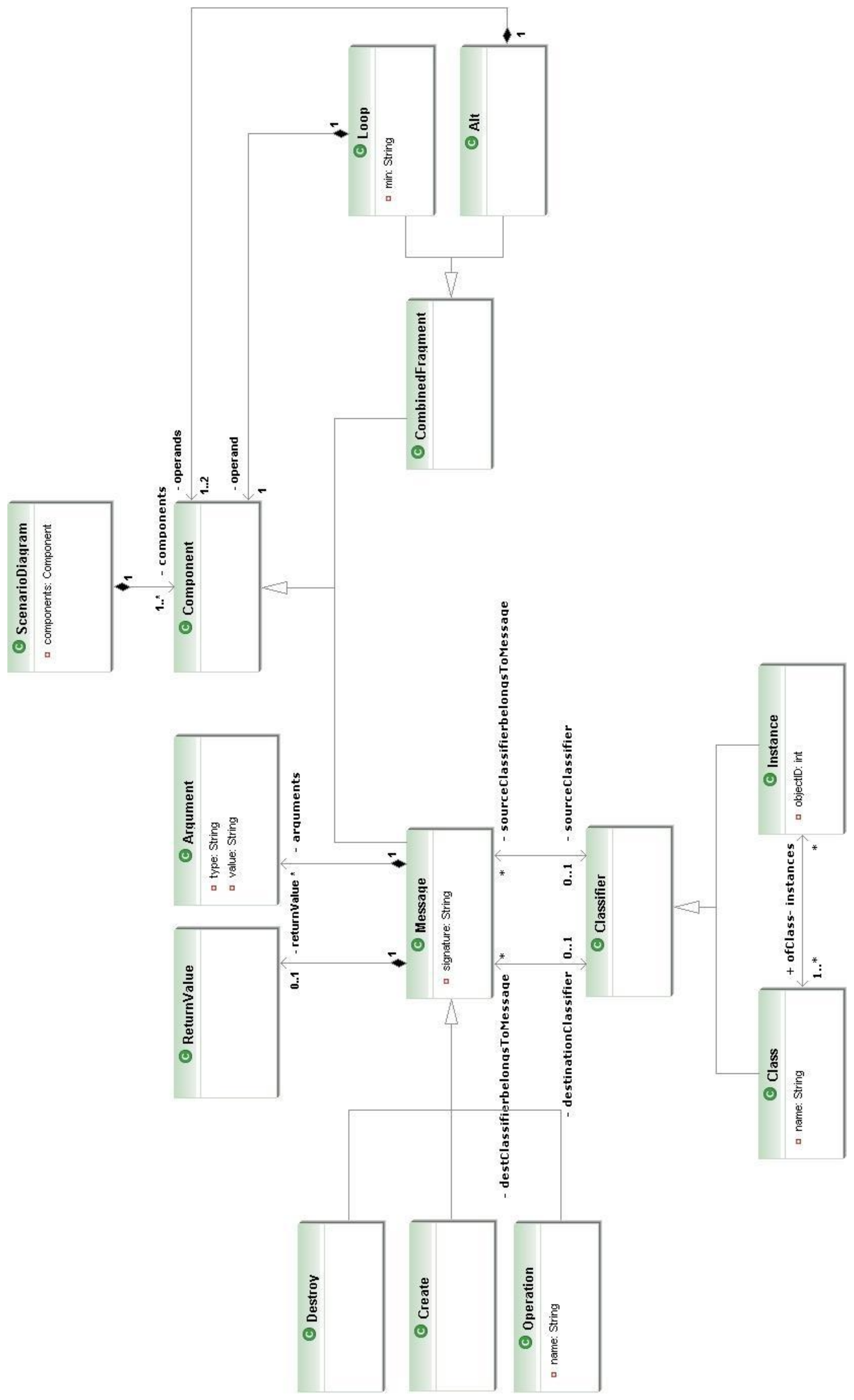


Figure 3.1: Scenario diagram metamodel

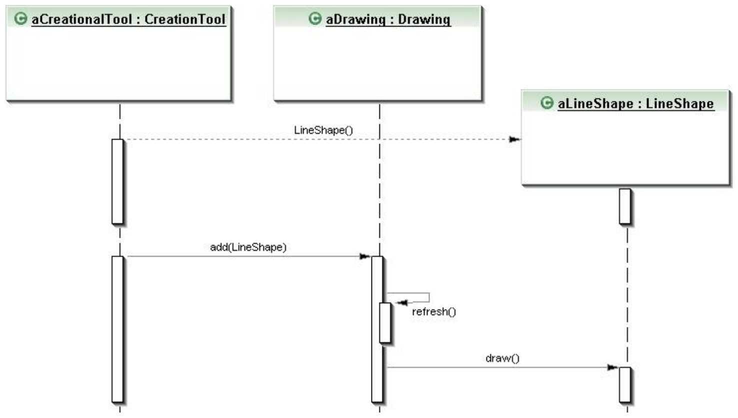


Figure 3.2: Scenario diagram notation

A vertical rectangle on a lifeline shows that the corresponding object is active; that is, it is handling a request. An object can send requests to other objects through method invocations; these are indicated with a horizontal arrow pointing to the receiving object. The name of the request is shown above the arrow. If an object send a message to itself, the message arrow points to the same lifeline.

Figure 3.2 is a simple scenario diagram that shows how a shape gets added to a drawing. It shows that the first request is issued by `aCreationTool` to create `aLineShape`. Later, `aLineShape` is `added` to `aDrawing`, which prompts `aDrawing` to send a `refresh` request to itself. Note that `aDrawing` sends a `draw` request to `aLineShape` as part of the `refresh` operation.

Following closely the described notation, we now consider the metamodel proposed by [BLL06], which captures the dynamic semantic concepts used to express the design solutions proposed by behavioral and creational patterns. This metamodel, illustrated in Figure 3.1, does not capture what a pattern is *in general*, but how it is used in one specific case, *i.e.*, its dynamic view.

A scenario diagram (class `ScenarioDiagram`) is composed of an ordered list of components (class `Component`) that can either be messages (class `Message`), or combined fragments (class `CombinedFragment`).

Messages can be of three different types: an operation call (class `Operation`), a destruction call (class `Destroy`), or a creation call (class `Create`). Messages have a `sourceClassifier` and a `destinationClassifier` to represent the concept of caller and callee. Caller and callee are of type `Classifier`, and are specialized as either an `Instance` or a `Class`, the latter applying to cases where the message in relation to the caller or callee is a class method. If appropriate, messages include arguments (class `Argument`) of different types: either data types or class types. The return value of messages is represented by class `ReturnValue`.

Class `CombinedFragment` is inspired by a previous notation [Obj04] to group sets of messages to show conditional flows in scenario diagrams. Although [Obj04] provides 11 interaction types of combined fragments, a thorough study of [GHJV94] shows that only the combined fragments *loop* and *alternative* are necessary to the identification of behavioral and creational design patterns. Consequently, combined fragments are specialized into two types: either loops (class `Loop`) to illustrate repetitions of messages, or alternatives (class `Alt`) to designate mutually exclusive choices among sequences of messages. To account for nested loops and alternatives, we introduce composition links: the composition relationships `operand` between classes `Loop` and `Component`, and between classes `Alt` and `Component`. A loop has one and only one operand, while an alternative has one or more operands. For instance, the classic alternative *if-then-else* has two operands: operand *if*, and operand *else* (our metamodel is extensible, and it is thus possible to add new constructs if new interaction types are needed in the future).

3.2 Modelling Behavioral and Creational Patterns

In [GHJV94], the authors expressed the elements that make up the collaboration in the solution advocated by a pattern in terms of a graphical notation similar to the aforementioned scenario diagram notation. Throughout this catalog of patterns, five out of the total 23 design patterns, *i.e.*, the Builder, Command, Memento, Observer, and Visitor patterns, are documented with this type of notation, which describes how the participants collaborate to carry out their responsibilities. Although only five patterns have such a graphical notation, our approach is not limited to their identification. Users can provide their own description of what they define as a pattern.

According to our approach (illustrated in Figure 1.1, Step 1), behavioral and creational design patterns are described by manually transforming the graphical notations of collaboration into an instance of the scenario diagram metamodel. In turn, this instance is used as the source system to identify design patterns in a target system, as explained in Chapter 5.

We now provide details of the aforementioned design patterns in terms of collaboration, and their translation into instances of the scenario diagram metamodel.

3.2.1 Builder

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. Figure 3.3 illustrates how `Builder` and `Director` cooperate with a client. The client creates the `Director` object and configures it with the desired `Builder` object. `Director` notifies the builder whenever a part of the product should be built. `Builder` handles requests from `Director` and adds parts to the product. The client retrieves the product from the builder.

For each message involved in the sequence of messages in Figure 3.3, *i.e.*, `ConcreteBuilder()`, `Director()`, `construct()`, `buildPartA()`, `buildPartB()`, `buildPartC()`, and `getResult()`, we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Builder pattern. The participants collaborating in the pattern, *i.e.*, `aClient`, `aDirector`, and `aConcreteBuilder`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of their corresponding messages. For example, the `sourceClassifier` of `ConcreteBuilder()` and `getResult()` is `aClient`, while `aConcreteBuilder` is their `destinationClassifier`.

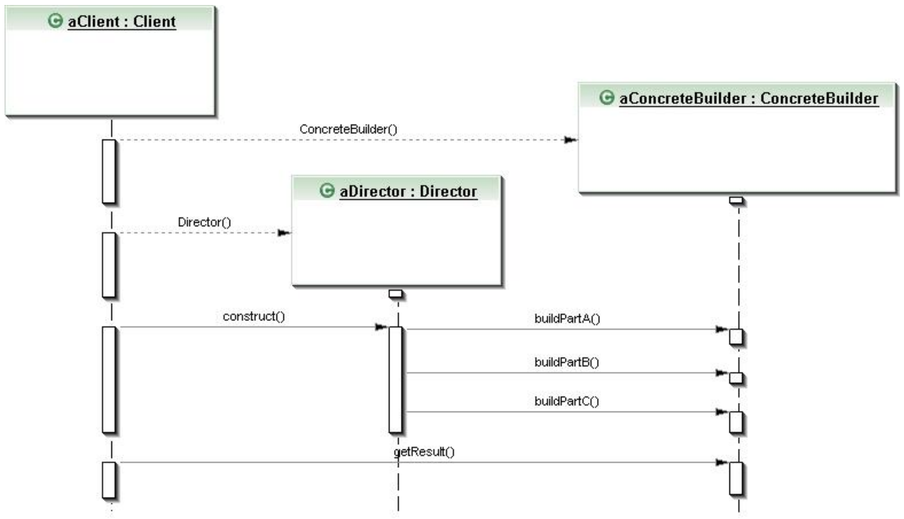


Figure 3.3: Description of the Builder pattern in terms of collaboration

3.2.2 Command

The Command pattern is most commonly used for encapsulating a request as an object, thereby parameterizing client with different requests, queues, or log requests, and support undoable operations. Figure 3.4 shows the interactions between the different collaborators. It illustrates how `Command` decouples the invoker from the receiver (and the request it carries out). The client creates a `ConcreteCommand` object and specifies its receiver. An `Invoker` object stores the `ConcreteCommand` object. The invoker issues a request by calling the `execute()` operation on the command. When commands are undoable, `ConcreteCommand` stores the state of the receiver for undoing the command prior to invoking `execute()`. The `ConcreteCommand` object invokes operations on its receiver to carry out the request.

For each message involved in the sequence of messages in Figure 3.4, *i.e.*, `Command(Receiver)`, `storeCommand(Command)`, `execute()`, and `action()`, we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Command pattern. Given message `setSubjectState(String)` takes an object of type `String` as argument, we in-

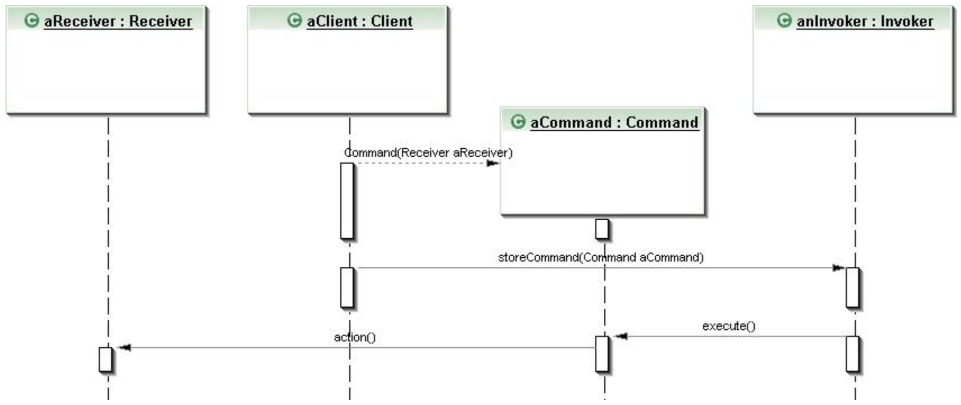


Figure 3.4: Description of the Command pattern in terms of collaboration

stantiate an object `Argument` which attribute `type` points to `String`. This object `Argument` is added to the ordered list `arguments` of message `setSubjectState(String)`. The participants collaborating in the pattern, `aConcreteSubject`, `aConcreteObserver`, and `anotherConcreteObserver`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of their corresponding messages. For example, the `sourceClassifier` of `setSubjectState(String)` and `getSubjectState()` is `aConcreteObserver`, while `aConcreteSubject` is their `destinationClassifier`.

3.2.3 Memento

Without violating encapsulation, the Memento pattern captures and externalizes an object's internal state so that the object can be restored to this state later. Figure 3.5 is a scenario diagram illustrating how the participants of this pattern collaborate. A caretaker requests a memento from an originator, holds it for a time, and passes it back to the originator. Sometimes the caretaker does not pass the memento back to the originator, because the originator might never need to revert to an earlier state. Mementos are passive. Only the originator that created a memento can assign or retrieve its state.

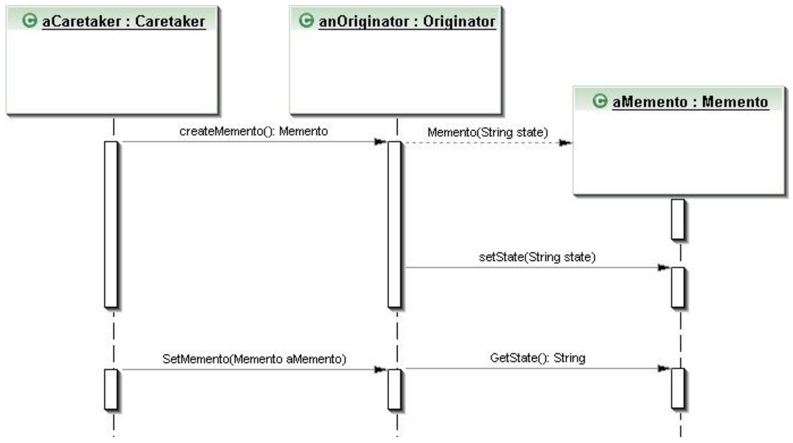


Figure 3.5: Description of the Memento pattern in terms of collaboration

For each message involved in the sequence of messages in Figure 3.5, *i.e.*, `createMemento()`, `Memento()`, `setState()`, `setMemento(Memento)`, and `getState()`, we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Memento pattern. Given message `setMemento(Memento)` takes an object of type `Memento` as argument, we instantiate an object `Argument` which attribute `type` points to `Memento`. This object `Argument` is added to the ordered list `arguments` of message `setMemento(Memento)`. The participants collaborating in the pattern, `aCaretaker`, `anOriginator`, and `aMemento`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of their corresponding messages. For example, the `sourceClassifier` of `createMemento()` and `setMemento(Memento)` is `aCaretaker`, while `anOriginator` is their `destinationClassifier`.

3.2.4 Observer

The Observer pattern defines a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically. The scenario diagram in Figure 3.6 illustrates the collaboration between a subject and two observers. A `ConcreteSubject` notifies its observers whenever a

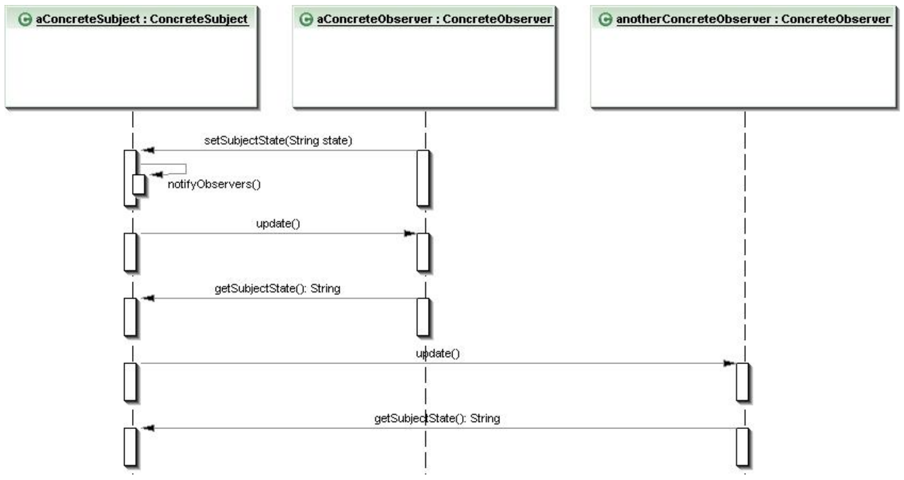

aConcreteSubject : ConcreteSubject
aConcreteObserver : ConcreteObserver
anotherConcreteObserver : ConcreteObserver
setSubjectState(String state)
notifyObservers()
update()
getSubjectState(): String
update()
getSubjectState(): String

Figure 3.6: Description of the Observer pattern in terms of collaboration

change occurs that could make its observers' state inconsistent with its own. After being informed of a change in the `ConcreteSubject`, a `ConcreteObserver` object may query the subject for information. `ConcreteObserver` uses this information to reconcile its state with that of the subject.

For each message involved in the sequence of messages in Figure 3.6, *i.e.*, `set-SubjectState()`, `notifyObservers()`, `update()`, `getSubjectState()`, `update()`, and `getSubjectState()`, we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Observer pattern. The participants collaborating in the pattern, `aConcreteSubject`, `aConcreteObserver`, and `anotherObserver`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of their corresponding messages. For example, the `sourceClassifier` of `setSubjectState()` and `getSujbectState()` is `aConcreteObserver`, while `aConcreteSubject` is their `destinationClassifier`.

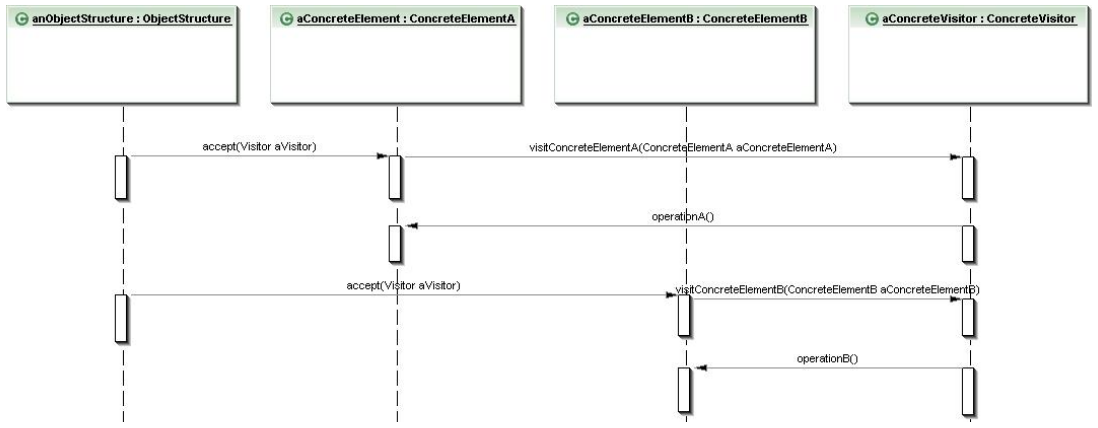

anObjectStructure : ObjectStructure
aConcreteElement : ConcreteElementA
aConcreteElementB : ConcreteElementB
aConcreteVisitor : ConcreteVisitor
accept(Visitor aVisitor)
visitConcreteElementA(ConcreteElementA aConcreteElementA)
operationA()
accept(Visitor aVisitor)
visitConcreteElementB(ConcreteElementB aConcreteElementB)
operationB()

Figure 3.7: Description of the Visitor pattern in terms of collaboration

3.2.5 Visitor

The Visitor pattern represents an operation to be performed on the elements of an object structure. Visitor allows defining a new operation without changing the classes of the elements on which it operates. Figure 3.7 illustrates the collaboration among an object structure, a visitor, and two elements. A client that uses the Visitor pattern creates a `ConcreteVisitor` object and then goes through the object structure, visiting each element using the visitor. When an element is visited, it calls the `visitConcreteElement(ConcreteElement)` operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

For each message involved in the sequence of messages in Figure 3.7, *i.e.*, `accept(Visitor)`, `visitConcreteElementA(ConcreteElement)`, `operationA()`, `accept(Visitor)`, `visitConcreteElementB(ConcreteElement)`, and `operationB()`, we instantiate an object `Operation` that is added to the ordered list `components` of an instance of `ScenarioDiagram` representing the Visitor pattern. Given message `accept(Visitor)` takes an object of type `Visitor` as argument, we instantiate an object `Argument` which attribute `type` points to `Visitor`. This object `Argument` is added to the ordered list `arguments` of message `accept(Visitor)`. The participants collaborating in the pattern, `anObjectStructure`, `aConcreteElementA`, `aConcreteElementB`, and `aConcreteVisitor`, are instantiated as instances of `Instance`, and are set to be the `sourceClassifier` or `destinationClassifier` of their corresponding messages. For example, the `sourceClassifier` of `visitConcreteElementA()` is `aConcreteElementA`, while `aConcreteVisitor` is its `destinationClassifier`.

3.3 Conclusion

In this chapter, we provided a way to formalize design patterns using a set of elements. These elements are defined in the core of a metamodel dedicated to the representation of patterns. The proposed metamodel provides means to describe behavioral aspects of design patterns. It establishes the conceptual machinery required to identify behavioral and creational design patterns in source code. Then, following the proposed metamodel, we described the Builder, Command, Memento, Observer, and Visitor patterns in terms of collaboration among objects as shown in [GHJV94], and showed how these collaboration is translated into instances of the scenario diagram metamodel.

In the next chapter, we will show how to reverse engineer a system's use case scenario into an instance of the same metamodel, with the view to perform the identification of some given design pattern collaborations (Chapter 5).

CHAPTER 4

REVERSE ENGINEERING OF SCENARIO DIAGRAMS

To extract scenario diagrams from software systems, we can choose to capture their behavior either by static analysis or dynamic analysis. Both strategies have their own advantages and limitations. On the one hand, even if static analysis can provide a complete picture of what can happen at runtime, it does not show what *actually* happens. Using static analysis to retrieve dynamic data requires to analyze the source code, and to determine the dynamic types of object references, which is not conceivable for large, complex systems [GJM91]. On the other hand, reverse engineered scenario diagrams obtained by dynamic analysis represent only *part* of the whole behavior of a system. Yet, they describe precisely the collaboration among objects.

In this thesis, we chose to use dynamic analysis because we favor precision over completeness. To cope with the incompleteness of the reverse engineered scenario diagrams, we shall consider in future work the merging of several execution traces, each reporting one observed behavior according to one scenario (or use case). The merging of scenario diagrams to generate a sequence diagram is clearly a difficult problem as stated in [HLBAL05], and calls for more research studying the synergy between static and dynamic approaches, similar to what has recently been suggested in [GZ05]. Also, the use of some test coverage tools could help in defining the scenarios that need to be executed to possibly recover all the design patterns applied during the design and implementation of a system.

4.1 Reverse Engineering Technique Using Dynamic Analysis

There are several techniques to retrieve data using dynamic analysis. These techniques include, for example, source code instrumentation [JSB97,RD99,BLM03], virtual machine instrumentation [WMFB$^+$98, PJM$^+$02], or the use of a customized debugger [SKM01, OS02]. In this thesis, we first tried to instrument the Java virtual machine Jikes RVM [Jik08]—Jikes Research Virtual Machine—which is designed to execute Java systems and to prototype new virtual machine technologies. However, the amount of time and work required to achieve successfully the necessary instrumentation quickly exceed the scope of this thesis. Therefore, we chose to use intermediate code instrumentation (bytecode instrumentation in the context of Java systems). To avoid having to maintain a "clean" version and an instrumented version of the same source code, and to avoid handling possible inconsistencies between the two versions, we choose to instrument the intermediate code, an operation which is less intrusive than virtual machine instrumentation and debugger customization. Nevertheless, some difficulties arise. First, delays due to the insertion of new instructions may be introduced while executing the instrumented system, with respect to the execution of the non-instrumented version. Yet, unless the instrumented system is a real-time system with hard synchronization constraints, the instrumentation should not change the intended behavior of the system. Also, there are threads and timing issues when the analyzed system is distributed. Generating a dynamic model showing distributed objects interactions, such as scenario diagrams, requires that messages be ordered, within or between threads execution on a computer, but also between threads execution on different computers. However, in a distributed system, there is often no global clock that could be used to order messages gathered from different computers. In the scope of this thesis, we only deal without concurrency. Therefore, the threads and timing issues are not relevant in the context of our instrumentation strategy.

The main limitation of the chosen strategy is its specificity to the target language. This strategy is tightly bonded to a particular runtime environment. However, the form and the contents of the dynamic data retrieved are not affected. Regardless of the language (as long as it is object-oriented) and the runtime environment of the system, a method execution is traced in such a way that its entry and exit are recorded in our approach.

In the literature, many approaches have been proposed to reverse engineer dynamic models of object-oriented systems using intermediate code instrumentation. Following [BLL06], we define a reverse engineering approach that consists in 5 steps (*cf.* Figure 1.1 on page 3, Steps 2i, 2ii, 2iii, 2iv, and 2v), adapted to systems written in Java. This approach could be generalized for any other object-oriented language, except for Step 2ii.

First, we compile the source files of a system to obtain their corresponding intermediate code. Second, we instrument the intermediate code. Third, we choose a scenario to be executed. The choice of the scenario is guided by the documentation of the system, where a list of functionalities is described, possibly including terms found in the description of design patterns (in the intent and motivation descriptions principally). Fourth, we execute the instrumented system following the chosen scenario to automatically produce an execution trace. Finally, we create an instance of the scenario diagram metamodel corresponding to the execution trace.

In the following, we illustrate the proposed instrumentation approach on a toy system that we implemented in Java according to the solution advocated by the Memento pattern in [GHJV94] (*cf.* Figure 3.5 on page 27). It will be used as a running example in Sections 4.1.1, 4.1.2, and 4.1.3. First, in Section 4.1.1, we present the execution trace format, which is required to achieve pattern identification. Our instrumentation strategy was developed to produce traces in this specific format, as explained in Section 4.1.2. Finally, we depict the instantiation of a scenario diagram from an execution trace (Step 2v) in Section 4.1.3.

4.1.1 Format of the Execution Trace

Each execution trace complies with the following syntax:

An execution trace contains a set of events. Each event corresponds to either the `start` or the `end` of a message. Each event also records the name of the message, its formal arguments, as well as the unique ID of the callee (receiver object). Traces are independent of the programming language, system, and application, and could be generated for any object-oriented programming language, in addition to Java presented here. The execution trace corresponding to the running example of the Memento pattern that complies with the described format is shown in Figure 4.1.

4.1.2 Instrumentation

Intermediate code instrumentation is specific to the target language and the runtime environment of the analyzed system. In this thesis, we chose to instrument Java bytecode generated by the widely used Java 2 Platform, Standard Edition version 1.4.2 with BCEL—the Byte Code Engineering Library [Apa06]. BCEL is a Java library that gives users the possibility to create, analyze, and easily manipulate Java class files. In this section, we will discuss the instrumentation strategy proposed to trace the execution of constructors and methods, as well as control flow and repetitions of messages.

4.1.2.1 Constructor and Method Executions

To identify design patterns in the scenario diagram of a system, we instrument constructor executions as well as method executions. Although requiring different means, the execution instrumentation of constructors is very similar to the one of methods. Therefore, we only present here the instrumentation of constructor executions.

$\langle \textit{trace}\rangle \rightarrow \langle \textit{event}\rangle \mid \langle \textit{event}\rangle\ \langle \textit{trace}\rangle$

$\langle \textit{event}\rangle \rightarrow \langle \textit{message\_type}\rangle\ \langle \textit{state}\rangle\ \langle \textit{message\_signature}\rangle$ **callee** $\langle \textit{class\_identifier}\rangle\ \{\langle \textit{id}\rangle\} \mid \langle \textit{repetition\_type}\rangle\ \langle \textit{state}\rangle$

$\langle \textit{message\_type}\rangle \rightarrow$ operation | constructor | destructor

$\langle \textit{repetition\_type}\rangle \rightarrow$ loop | alt

$\langle \textit{state}\rangle \rightarrow$ start | end

$\langle \textit{message\_signature}\rangle \rightarrow \{\langle \textit{visibility}\rangle\}$ {**static**} $\langle \textit{return\_type}\rangle\ \langle \textit{message\_identifier}\rangle$ **(** $\{\langle \textit{argument}\rangle\}$ **)**

$\langle \textit{visibility}\rangle \rightarrow$ public | private | protected

$\langle \textit{message\_identifier}\rangle \rightarrow$ *name of the message being called*

$\langle \textit{argument}\rangle \rightarrow \langle \textit{argument\_type}\rangle\ \langle \textit{argument\_identifier}\rangle\ \{,\ \langle \textit{argument}\rangle\}$

$\langle \textit{argument\_type}\rangle \rightarrow$ *type of the argument*

$\langle \textit{argument\_identifier}\rangle \rightarrow$ *name of the argument*

$\langle \textit{class\_identifier}\rangle \rightarrow$ *name of the class to which belongs the message being called*

$\langle \textit{id}\rangle \rightarrow$ *unique number of the instance object to which belongs the message being called*

```
 1:   operation start public static void main(String[] args) callee ModelMementoTest -1
 2:   constructor start public void <init>() callee Caretaker 14613018
 3:   constructor start public void <init>() callee Originator 12386568
 4:   constructor end  public void <init>() callee Originator 12386568
 5:   constructor end public void <init>() callee Caretaker 14613018
 6:   operation start public void callCreateMemento() callee Caretaker 14613018
 7:   operation start public Memento createMemento() callee Originator 12386568
 8:   constructor start public void <init>() callee Memento 17237886
 9:   constructor end public void <init>() callee Memento 17237886
10:   operation start public void setState(String state) callee Memento 17237886
11:   operation end public void setState(String state) callee Memento 17237886
12:   operation end public Memento createMemento() callee Originator 12386568
13:   operation end public void callCreateMemento() callee Caretaker 14613018
14:   operation start public void undoOperation() callee Caretaker 14613018
15:   operation start public void setMemento(Memento m) callee Originator 12386568
16:   operation start public String getState() callee Memento 17237886
17:   operation end public String getState() callee Memento 17237886
18:   operation end public void setMemento(Memento m) callee Originator 12386568
19:   operation end public void undoOperation() callee Caretaker 14613018
20:   operation end void public static void main (String[] args) callee ModelMementoTest -1
```

Figure 4.1: Execution trace of a toy system implementing the Memento pattern

In the Java Virtual Machine, every constructor appears as an instance initialization method that has the special name `<init>`. Figure 4.2 illustrates the constructor of class `Memento` from the Memento pattern shown in Figure 3.5 on page 27. Its bytecode instructions before instrumentation are shown in Figure 4.3. To produce dynamic data when constructors start executing, we insert bytecode instructions after the invocation of the superclass constructor, to enforce the writing of a `constructor start` event in the execution trace. These bytecode instructions are illustrated by the lines 3–5 in Figure 4.4. For tracing the corresponding `constructor end` event, we identify every `return` bytecode instruction and every call to `System.exit(int)` (the only way in Java to exit a method *normally*). Then, before each such instruction, we insert bytecode instructions to write a `constructor end` event in the execution trace, as illustrated by the lines 19–21 in Figure 4.4. Each time a `constructor start` event or `constructor end` event is traced, bytecode instructions to trace a constructor `destinationClassifier` class name and identity code are also added to the original instructions. To uniquely identify each instance object in the system, we use the object identity hash-code. These

```
public class Memento {

   public Memento (String state) {
        this.setState(state);
   }
}
```

Figure 4.2: Partial source code of class Memento

```
1: aload_0
2: invokespecial Object.<init> ()V
3: aload_0
4: ldc           "state"
5: putfield      Originator.state String;
6: return
```

Figure 4.3: Constructor bytecode before instrumentation

```
 1:   aload_0
 2:   invokespecial Object.<init> ()V
 3:   ldc           "Memento_GOF_EVALUATION.trace"
 4:   ldc           "constructor start <init> CALLEE Originator"
 5:   invokestatic  LogToFile.write (Object;)V
 6:   ldc           "Memento_GOF_EVALUATION.trace"
 7:   new           <Integer>
 8:   dup
 9:   aload_0
10:   invokestatic  System.identityHashCode (Object;)I
11:   invokespecial Integer.<init> (I)V
12:   invokestatic  LogToFile.write (Object;)V
13:   ldc           "Memento_GOF_EVALUATION.trace"
14:   ldc           "\n"
15:   invokestatic  LogToFile.write (Object;)V
16:   aload_0
17:   ldc           "state"
18:   putfield      Originator.state Ljava/lang/String;
19:   ldc           "Memento_GOF_EVALUATION.trace"
20:   ldc           "constructor end <init> CALLEE Originator"
21:   invokestatic  LogToFile.write (Object;)V
22:   ldc           "Memento_GOF_EVALUATION.trace"
23:   new           <Integer>
24:   dup
25:   aload_0
26:   invokestatic  System.identityHashCode(Object;)I
27:   invokespecial Integer.<init> (I)V
28:   invokestatic  LogToFile.write(Object;)V
29:   ldc           "Memento_GOF_EVALUATION.trace"
30:   ldc           "\n"
31:   invokestatic  LogToFile.write(Object;)V
32:   return
```

Figure 4.4: Constructor bytecode after instrumentation

bytecode instructions are illustrated by the lines 10–15 and 26–31 in Figure 4.4. The `sourceClassifier` of a constructor is determined only while instantiating the scenario diagram, as discussed in Section 4.1.3. Multithreading has not been an issue so far for the design pattern identification, so we do not distinguish threads.

4.1.2.2 Control Flow and Repetitions

The instrumentation strategy produces an execution trace that provides all the necessary data to generate scenario diagrams to perform design pattern identification. The data may include the conditions corresponding to the flow of control, and the repetitions of messages—referred to as `loop start` and `loop end` in our execution trace—which are used to instantiate class `Loop` in the scenario diagram metamodel.

Bcel allows users to manipulate Java bytecode easily, so we have also chosen to instrument the methods bytecode to obtain control-flow structures, instead of discovering patterns of executions in the execution trace or the source code, as in previous work [DHKV93, JSB97, RD99, SKM01]. Tracing conditions is similar to tracing repetitions of messages, so we only describe the latter.

We produce dynamic data indicating when loops start and end. Every message that appears between dynamic data `loop start` and `loop end` is considered as a message called in the loop. Figure 4.5 depicts an example of source code involving repetition of message `m1()`, which is positioned inside the loops `for` (line 11) and `while` (line 14). Before instrumentation, the bytecode instructions of method `main(String[])` are shown in Figure 4.6. Our approach to instrument `loop start` and `loop end` is as follows. Given a method of a class, we locate bytecode instructions specific to control-flow structures, such as `for` and `while`.

To trace a `loop start`, we identify every branch instruction whose target is indexed before its own position. For instance, in Figure 4.6, branch instructions at lines 19 and 25 have targets that are positioned at an index inferior to their positions, respectively at lines 8 and 3. Then, previous to the instruction in relation

```
public class TestRepetition {

   public static void m1 ()
   {
      a++;
   }

   public static void main (String args [])
   {
      int counter;
      for(counter = 0; counter < 5; counter++)
      {
           a = 0;
           while (a < 2)
           {
              if(counter == 1)
                 break ;
              else if (a < 2)
                 m1();
           }
           if(counter == 2)
              return ;
      }
   }

}
```

Figure 4.5: Source code of class `TestRepetition`.

```
 1: iconst_0
 2: istore_1
 3: iload_1
 4: iconst_5
 5: if_icmpge     #26
 6: iconst_0
 7: putstatic     Loops.a I
 8: getstatic     Loops.a I
 9: iconst_2
10: if_icmpge     #20
11: iload_1
12: iconst_1
13: if_icmpne     #15
14: goto          #20
15: getstatic     Loops.a I
16: iconst_2
17: if_icmpge     #8
18: invokestatic  Loops.m1 ()V
19: goto          #8
20: iload_1
21: iconst_2
22: if_icmpne     #24
23: return
24: iinc          %1  1
25: goto          #3
26: return
```

Figure 4.6: Bytecode of method `main(String[])` before instrumentation

to the branch instruction target, we insert bytecode instructions to write a `loop start` in the execution trace, as shown in Figure 4.7 at lines 3–6.

Tracing a `loop end` requires that we locate every bytecode instruction that may cause a loop iteration to end. In particular, in the range of a loop scope, we locate every `if` or `goto` bytecode instruction whose target's index is positioned outside the loop scope. Then, we insert before the corresponding target instruction, bytecode instructions to write a `loop end`. For example, in Figure 4.6, bytecode instruction at line 5 is an `if` instruction positioned inside the loop defined at line 25. In addition, the `if` target instruction, positioned at line 26, is outside the loop scope. Therefore, we insert dynamic data before line 26 to indicate a `loop end`. The corresponding bytecode newly inserted is shown in Figure 4.7 at lines 50–53. As a result, if an iteration of the loop starts and ends, the corresponding notifications appear in the execution trace.

```
 1:   iconst_0
 2:   istore_1
 3:   ldc           "ControlFlow.trace"
 4:   ldc           "loop start #1\n"
 5:   invokestatic  LogToFile.write(String;Object;)V
 6:   iload_1
 7:   iconst_5
 8:   if_icmpge     #50
 9:   iconst_0
10:   putstatic     Loops.a I
11:   ldc           "ControlFlow.trace"
12:   ldc           "loop start #2\n"
13:   invokestatic  LogToFile.write(String;Object;)V
14:   getstatic     Loops.a I
15:   iconst_2
16:   if_icmpge     #32
17:   iload_1
18:   iconst_1
19:   if_icmpne     #23
20:   goto          #32
21:   getstatic     Loops.a I
22:   iconst_2
23:   if_icmpge     #12
24:   invokestatic  Loops.m1 ()V
25:   ldc           "ControlFlow.trace"
26:   ldc           "loop end #2\n"
27:   invokestatic  LogToFile.write(String;Object;)V
28:   goto          #12
29:   ldc           "ControlFlow.trace"
30:   ldc           "loop end #2\n"
31:   invokestatic  LogToFile.write(String;Object;)V
32:   iload_1
33:   iconst_2
34:   if_icmpne     #44
35:   ldc           "ControlFlow.trace"
36:   ldc           "all loops exit\n"
37:   invokestatic  LogToFile.write(String;Object;)V
38:   return
39:   iinc          %1  1
40:   ldc           "ControlFlow.trace"
41:   ldc           "loop end #1\n"
42:   invokestatic  LogToFile.write(String;Object;)V
43:   goto          #3
44:   ldc           "ControlFlow.trace"
45:   ldc           "loop end #1\n"
46:   invokestatic  LogToFile.write(String;Object;)V
47:   return
```

Figure 4.7: Bytecode of method `main(String[])` after instrumentation

```
<OPERATION> public static void main (String[] args)
        <CALLEE> ModelMementoTest       <CALLER> nonexistent
<CREATE> public void <init>()
        <CALLEE> Caretaker 14613018     <CALLER> ModelMementoTest
<CREATE> public void <init>()
        <CALLEE> Originator 12386568    <CALLER> Caretaker 14613018
<OPERATION> public void callCreateMemento()
        <CALLEE> Caretaker 14613018     <CALLER> ModelMementoTest
<OPERATION> public Memento createMemento()
        <CALLEE> Originator 12386568    <CALLER> Caretaker 14613018
<CREATE> public void <init>()
        <CALLEE> Memento 17237886       <CALLER> Originator 12386568
<OPERATION> public void setState(String state)
        <CALLEE> Memento 17237886       <CALLER> Originator 12386568
<OPERATION> public void undoOperation()
        <CALLEE> Caretaker 14613018     <CALLER> ModelMementoTest
<OPERATION> public void setMemento(Memento m)
        <CALLEE> Originator 12386568    <CALLER> Caretaker 14613018
<OPERATION> public String getState()
        <CALLEE> Memento 17237886       <CALLER> Originator 12386568
```

Figure 4.8: Textual representation of the scenario diagram after that the execution trace of Figure 4.1 has been processed

4.1.3 Instantiation of Scenario Diagram

After the execution of an instrumented system, we obtain an *incomplete* execution trace for one use case. There is one important data that was left out during the instrumentation: the caller of each message (*i.e.*, sender object). We identify the `sourceClassifier` of a message by determining the callee (receiver) of the occurrence of an event in the execution trace positioned just before the currently analyzed message, and specifying a message execution start. In Figure 4.1, the preceding message of `Message createMemento()` is `Message callCreateMemento()` at line 6. The `sourceClassifier` of `createMemento()` is then set as the `destinationClassifier` of `callCreateMemento()`.

Once the `sourceClassifier` of each message is identified, the execution trace is processed to determine the corresponding scenario diagram. This process is independent of the target language of the system, as long as the execution trace has the aforementioned format.

Figure 4.8 is a textual representation of the scenario diagram corresponding to the execution trace in Figure 4.1. For each execution trace event such as `operation start` or `constructor start`, a message of type `Operation` or `Create` is respectively instantiated[1], while an object `CombinedFragment` of type `Loop` or `Alt` is instantiated for each execution trace event `loop start` or `alt start`. In both cases, the component corresponding to the event currently analyzed in the execution trace is referred to as the *current component*. If the current component is of type `CombinedFragment`, we add the subsequent objects `Message` or `CombinedFragment` to its ordered list `operands`, until the corresponding `end` event is met. Otherwise, they are added to the ordered list `components` of object `ScenarioDiagram`. Each time an object `Message` is instantiated, its corresponding `sourceClassifier` and `destinationClassifier` of type `Classifier` are also instantiated (if needed). The set `arguments` of a message is determined by processing the data positioned between the parentheses of the corresponding event.

4.2 Conclusion

In this chapter, we showed how to instrument intermediate code to obtain the order in which constructors and methods are executed, and control flow data (in particular, the repetition of messages) for a given use case of a Java system. In addition, we explained how to instantiate the scenario diagram metamodel from the execution trace produced by the execution of an instrumented system.

We will now perform the identification of behavioral and creational design patterns using two instances of the scenario diagram metamodel, one resulting from the description of the searched design pattern (*i.e.*, the source system), and the other, from the execution of some use case of a system (*i.e.*, the target system).

[1]Explicitly, we do not refer to type `Destroy` here because of the nature of Java, as Java has no destructor or similar concept. However, the scenario diagram instantiation process is not limited to Java, since the format of an execution trace complies with other object-oriented programming languages.

CHAPTER 5

IDENTIFICATION OF DESIGN PATTERNS

Using the process and techniques described in the previous chapters, we obtained two scenario diagrams: one instance models the sequence of messages of the searched design pattern (*i.e.*, the source system), while the other models the sequence of messages for the executed scenario (*i.e.*, the target system). The approach we follow to identify behavioral and creational design patterns consists in identifying occurrences of the scenario diagrams of the source system in the scenario diagram of the target system, *i.e.*, in matching exactly the collaboration of the searched design pattern with those of the executed scenario of a software system. As illustrated in Figure 1.1, Step 3, we translate this identification process into an eCSP, as in previous work [GAA01], in terms of variables, constraints among variables, and domains of the variables. The eCSP represents the problem that the explanation-based constraint solver JCHOCO [JB00] solves to identify in the target system, sequences of messages that are identical or similar to the one defined in the searched design pattern.

As in the previous chapter, we will illustrate the translation of the identification process into an eCSP using the running example that we implemented following the solution of the Memento pattern in [GHJV94] (*cf.* Figure 3.5 on page 27).

5.1 Explanation-based Constraint-Programming

Explaining and suggesting possible architectural modifications is an interesting way to improve object-oriented source code. These explanations can not only ease program comprehension, but also help improve the quality of the system. Explanation-based constraint programming already proved to be of interest in many applications [JB00].

5.1.1 Explanations

In the following, we consider a CSP (V, D, C). Decisions made during the enumeration phase (variable assignments) correspond to adding or removing constraints from the current constraint system (*e.g.*, upon backtracking).

A contradiction explanation (*a.k.a.* nogood [SV94]) is a subset of the current constraint system of the problem that, left alone, leads to a contradiction (no feasible solution contains a nogood). A contradiction explanation is divided into two parts: a subset of the original set of constraints ($C' \subset C$ in Equation 1) and a subset of decision constraints introduced so far in the search.

$$C \vdash \neg(C' \wedge v_1 = a_1 \wedge ... \wedge v_k = a_k) \tag{1}$$

In a contradiction explanation composed of at least one decision constraint, a variable v_j is selected and the previous formula is rewritten as[2]:

$$C \vdash C' \wedge \bigwedge_{i \in [1..k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j \tag{2}$$

The left hand side of the implication constitutes an eliminating explanation for the removal of value a_j from the domain of variable v_j and is noted $\texttt{expl}(v_j \neq a_j)$.

Classical CSP solvers use domain-reduction techniques (removal of values). Recording eliminating explanations is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the domain of a variable v_j is emptied. A contradiction explanation can easily be computed with the eliminating explanations associated with each removed value:

$$C \vdash \neg\left(\bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a)\right) \tag{3}$$

[2]A contradiction explanation that does not contain such a constraint denotes an over-constrained problem.

There exist generally several eliminating explanations for the removal of a given value. Recording all of them leads to an exponential space complexity. Another technique relies on forgetting (erasing) eliminating explanations that are no longer relevant[3] to the current variable assignment. This way, the space complexity remains polynomial. We keep only one explanation at a time for a value removal.

5.1.2 Computing Explanations

Minimal (*w.r.t.* inclusion) explanations are the most interesting. They provide very precise details about possible dependencies among variables and constraints that are identified during the search. Unfortunately, computing such explanations is time-consuming [Jun01]. A good compromise between size and computability is the use of the knowledge that is inside the solver: constraint solvers always know (although not often explicitly) why values are removed from the domain of variables. Precise and interesting eliminating explanations can be calculated by explicitly stating such information.

5.1.3 Using Explanations

Explanations can be used in several ways [JDB00, JB00, JL00]. Debugging purposes are the most obvious: to explain clearly failures, to explain differences between intended and observed behavior for a given problem.

Explanations can also be used to determine direct or indirect effects of a given constraint on the domain of the variables of the problem, and for dynamic constraint removal. This is the case with the justification system used in [Bes91] for solving dynamic CSP. This justification system is actually a partial explanation system. Moreover, being able to explain failure and to dynamically remove a constraint facilitate the building of dynamic over-constrained problem solver [JB97].

[3]A nogood is said to be relevant if all the decision constraints in it are still valid in the current search state [BM96].

Other less direct applications exist as well: in particular, using explanation to guide the search. Indeed, classical backtracking-based searches only proceed when encountering failures (by backtracking to the last choice point). Contradiction explanation can be used in several ways to improve standard backtracking and to exploit data gathered to improve the search, by providing intelligent backtracking [GJP00], replacing standard backtracking with a jump-based approach *à la Dynamic Backtracking* [Gin93, JDB00], or even developing new local searches on partial instantiations [JL00].

In the context of design pattern identification, what is most interesting in using explanation systems is the ability to:

- Explain why no solution is found for a given problem. As stated before, a contradiction explanation that does not contain any decision constraints denotes an over-constrained system (*i.e.*, a system with no possible solutions). Such explanations are recursively obtained after having tested all possible values for a given variable [JB00].

- Provide insights on the available variants/approximations of patterns (*i.e.*, on the constraint relaxations that would lead to more occurrences, if the associated constraints were relaxed): a contradiction explanation justifies why there are not more solutions for the current problem. Selecting and relaxing a constraint given by the explanation allows the discovery of new solutions (approximate solutions for the original problem). In this thesis, the selection is left to the developer who knows which constraint to relax, without deviating from behavioral specification of the design pattern being searched for. However, in future work, we will consider allowing the user to give his inputs regarding the constraints suggested by the solver that should be removed from the system.

5.2 Application to the Problem of Design Pattern Identification

The identification of behavioral and creational design patterns using explanation-based constraint programming consists:

1. In modelling a set of design patterns as a set of eCSPs: the dynamic solution advocated by a design pattern is modelled as a set of constraints. A variable is associated with each participant in the scenario diagram defined for a design pattern. The collaboration among classes (caller/callee, messages order, etc.) is represented by constraints between the variables.

2. In modelling the execution of a scenario of the target system, to keep only the data needed to apply the constraints: the messages sent during the execution of the chosen scenario, and the caller/callee of each message are modelled.

3. In solving the eCSP to search for exact and *approximate* solutions—solutions for which one or several constraints specified by the design pattern are relaxed: when all the complete solutions of the eCSP are found, the search is dynamically guided by the user to find interesting approximate solutions. Information (explanations of contradiction) provided by the constraint solver helps the user.

More precisely, in our design pattern identification approach based on the scenario diagram metamodel proposed in Chapter 3, the different constituents of the eCSP are defined as follows:

Variables. The set of variables of the eCSP corresponds to the scenario diagram metamodel instances of `Classifier` and `Message` modelling the scenario diagram of a design pattern (the source system). The variables of our model are integer-valued.

Domain. The domain of each variable of the eCSP corresponds to a set of integers, each corresponding to a scenario diagram metamodel instance of `Classifier` or `Message` in the scenario diagram of a target system. Each instance of `Classifier` or `Message` is identified by a unique integer.

Constraints. The set of constraints among the variables of the eCSP corresponds to the relation expressed among the scenario diagram metamodel instances of `Classifier` or `Message` contained in the scenario diagram of a design pattern. We use a binary constraint in the form `constraint(variable1, variable2)` to express the existence of a collaboration between `variable1` and `variable2`. It is worth noting the eCSP that we define can involve constraints among variables of different entities in the scenario diagram metamodel, as opposed to previous work using eCSP, which allows only the definition of constraints among the same type of entities. For instance, in this thesis, these variables denote `Classifier` instances or `Message` instances.

In the case of the running example, the Memento pattern shown in Figure 3.5 on page 27 is the source system. The corresponding eCSP is expressed by associating a variable with each of its instances of `Message` (`var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState`), and its instances of `Classifier` (`var_aCaretaker`, `var_anOriginator`, and `var_aMemento`).

The domain of each variable of the eCSP corresponds to the scenario diagram metamodel instances of `Classifier` or `Message` in the target system. In the case of the running example, the scenario diagram of the target system in which we want to identify the Memento pattern is shown in Figure 4.8, and comprises ten instances of `Message` (`main (String[])`, `public void <init>()`, `public void <init>()`, `public void callCreateMemento()`, `createMemento()`, `public void <init>()`, `setState()`, `undoOperation()`, `setMemento(aMemento)`, and `getState()`), and four instances of `Classifier` (`ModelMementoTest`, `Caretaker`, `Originator`, and Memento). Therefore, the domain of variables `var_aCaretaker`, `var_anOriginator`, `var_aMemento`, and variables `var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState` are respectively each of size four and ten.

5.2.1 A Library of Specialized Constraints

From the collaboration among classes defined in [GHJV94], we built a library of constraints. Specialized constraints express the caller/callee, messages order, etc. relationships. These constraints involve variables representing one and only one class or message, because the tools we use do not manage (yet) constraints on sets. We use a simple trick to handle constraints on sets: variables representing sets of classes are not enumerated during the problem solving.

To the best of our knowledge, in previous work in which CSP is used to identify design patterns, constraints are only defined among variables of the same type, for example [QYW97] or [GAA01]. In our approach, constraints can be defined among variables of different types (`Classifier` and–or `Message`). It leads to a greater precision while describing a design pattern because a relationship can be expressed among instances of `Message`, `Classifier`, or between an instance of `Message` and an instance of `Classifier`.

The set of constraints used in our approach to express the relations between variables, which can be combined to form more complex constraints, includes:

Constraint `caller (classifier1, message2)` (respectively `callee`) defines the relation *`classifier1` is the `sourceClassifer` of `message2`* (respectively *`classifier1` is the `destinationClassifier` of `message2`*) between `classifier1` and `message2`. As shown in Algorithm 1, the domain of variable `classifier1` corresponds to the instances of `Classifier` in the target system. The domain of variable `message2` corresponds to the instances of `Message` in the target system. For each value taken by `message2`, there must be a corresponding value taken by `classifier1` so that `classifier1` is the `sourceClassifier` of `message2`. Conversely, for each possible value taken by `classifier1`, there must be a corresponding value taken by `message2` so

Algorithm 1 Constraint `caller(classifier1, message2)` (respectively `callee`)

```
1: toBeRemoved ← true
2: domain1 ← Domain(classifier1)
3: for i = 0 to Size(domain1) and toBeRemoved = true do
4:    classifier ← ElementAt(domain1, i)
5:    listOfMessages ← MessagesWhoseCallerIs(classifier)
6:    for j = 0 to Size(listOfMessages) and toBeRemoved = true do
7:       aMessage ← ElementAt(listOfMessages, j)
8:       domain2 ← Domain(message2)
9:       if ContainsMessage(domain2, message2) then
10:         toBeRemoved ← false
11:      end if
12:   end for
13: end for
14: toBeRemoved ← true
15: domain2 ← Domain(message2)
16: for i = 0 to Size(domain2) and toBeRemoved = true do
17:    message ← ElementAt(domain2, i)
18:    caller ← CallerOf(message)
19:    if ContainsClassifier(domain1, caller) then
20:       toBeRemoved ← false
21:    end if
22: end for
```

that the `sourceClassifier` of `message2` is a `Classifier` in the domain of `classifier1`. Any value of `classifier1` and `message2` failing to comply to this collaboration is removed from the corresponding domain.

Constraint `creator(classifier1, message2)` (respectively `created`) is similar to constraint `caller(classifier1, message2)`, except that `message2` is an instance of `Create` instead of `Operation` (*cf.* Figure 3.1 on page 21). For each possible value of `message2`, there must be a corresponding value of `classifier1` so that `classifier1` is an instance of `Create`, and is the `sourceClassifier` of `message2`

Constraint `notEqual(classifier1, classfier2)` (respectively `(message1, message2)`) defines the relation ***`classifier`** is not equal to **`classifier2`*** (respectively ***`message1`** is not equal to **`message2`***).

Constraint `follows(message1, message2)` defines the relation ***`message2`** is executed after **`message1`***. The domain of variables `message1` and `message2` corresponds to the instances of `Message` in the scenario diagram of the target system. For each possible value taken by `message2`, there must be a corresponding value taken by `message1` so that `message2` is called after `message1`. Conversely, for each possible value taken by `message1`, there must be a corresponding value taken by `message2` so that `message1` is called before `message2`. Any value of `message1` and `message2` failing to comply to this collaboration is removed from the corresponding domain.

Constraint `parameterCalleeSameType(message1, message2)` defines the relation *Parameter of **`message1`**, if any, is of same type as the callee of **`message2`***. The domain of variables `message1` and `message2` corresponds to the instances of `Message` in the scenario diagram of the target system. For each possible value taken by `message2`, there must be a corresponding value taken by `message1` so that the callee of `message2` is of same type as a parameter of `message1`. Conversely, for each possible value taken by `message1`,

Algorithm 2 Constraint `creator(classifier1, message2)` (respectively `created`)

```
1: toBeRemoved ← true
2: domain1 ← Domain(classifier1)
3: for i = 0 to Size(domain1) and toBeRemoved = true do
4:    classifier ← ElementAt(domain1, i)
5:    listOfMessages ← MessagesWhoseCallerIs(classifier)
6:    for j = 0 to Size(listOfMessages) and toBeRemoved = true do
7:       aMessage ← ElementAt(listOfMessages, j)
8:       domain2 ← Domain(message2)
9:       if ContainsMessage(domain2, message2) and IsOfType(aMessage,“Create”)
         then
10:         toBeRemoved ← false
11:      end if
12:   end for
13: end for
14: toBeRemoved ← true
15: domain2 ← Domain(message2)
16: for i = 0 to Size(domain2) and toBeRemoved = true do
17:    message ← ElementAt(domain2, i)
18:    caller ← CallerOf(message)
19:    if ContainsClassifier(domain1, caller) and IsOfType(message2, “Create”) then
20:       toBeRemoved ← false
21:    end if
22: end for
```

Algorithm 3 Constraint `follows(message1, message2)`

```
1: toBeRemoved ← true
2: domain1 ← Domain(message1)
3: for i = 0 to Size(domain1) and toBeRemoved = true do
4:    aMessage1 ← ElementAt(domain1, i)
5:    if IsCalledBeforeEveryMessageOf(aMessage1, domain2) then
6:       toBeRemoved ← false
7:    end if
8: end for
9: toBeRemoved ← true
10: domain2 ← Domain(message2)
11: for $i = 0$ $to$ $Size(domain2)$ $and$ $toBeRemoved = true$ do
12:    aMessage2 ← ElementAt(domain2, i)
13:    if IsCalledBeforeEveryMessageOf(aMessage2, domain1) then
14:       toBeRemoved ← false
15:    end if
16: end for
```

Algorithm 4 Constraint `parameterCalleeSameType(message1, message2)`

```
1: toBeRemoved ← true
2: domain1 ← Domain(message1)
3: domain2 ← Domain(message2)
4: for i = 0 to Size(domain1) and toBeRemoved = true do
5:    aMessage1 ← ElementAt(domain1, i)
6:    for j = 0 to Size(domain2) do
7:       aMessage2 ← ElementAt(domain2, j)
8:       callee ← CalleeOf(aMessage2)
9:       for each argument of aMessage1 and toBeRemoved = true do
10:         if IsOfType(argument, TypeOf(callee)) then
11:            toBeRemoved ← false
12:         end if
13:      end for
14:   end for
15: end for
16: toBeRemoved ← true
17: for $i = 0$ $to$ $Size(domain2)$ $and$ $toBeRemoved = true$ do
18:   aMessage2 ← ElementAt(domain2, i)
19:   for j = 0 to Size(domain1) do
20:      aMessage1 ← ElementAt(domain1, j)
21:      callee ← CalleeOf(aMessage2)
22:      for each argument of aMessage1 and toBeRemoved = true do
23:         if IsOfType(argument, TypeOf(callee)) then
24:            toBeRemoved ← false
25:         end if
26:      end for
27:   end for
28: end for
```

there must be a corresponding value taken by `message2` so that the parameter of `message1` is of same type as the caller of `message2`. Each value of `message1` and `message2` failing to comply to this collaboration are removed from the corresponding domains.

In the case of the running example, the collaboration among the entities of the scenario diagram are translated into constraints as:

```
 1. follows(var_createMemento, var_newMemento)
 2. follows(var_newMemento,var_setState)
 3. follows(var_setState, var_setMemento)
 4. follows(var_setMemento, var_getState)
 5. caller(var_aCaretaker, var_createMemento)
 6. callee(var_anOriginator, var_createMemento)
 7. creator(var_anOriginator, var_newMemento)
 8. created(var_aMemento, var_newMemento)
 9. caller(var_anOriginator, var_setState)
10. callee(var_aMemento, var_setState)
11. caller(var_aCaretaker, var_setMemento)
12. callee(var_anOriginator, var_setMemento)
13. caller(var_anOriginator, var_getState)
14. callee(var_aMemento, var_getState)
```

5.2.2 Solver

Explanation-based constraint programming [Jus01] is the key tool for identifying complete and approximate solutions without having to describe all possible variants, as shown in previous work [GAA01]. First, complete occurrences are computed. This computation ends by a contradiction (there are no more occurrences). Explanation-based constraint programming provides a contradiction explanation: the set of constraints justifying that any other combination of entities do not verify the constraints describing the design pattern. A contradiction explanation provides insights on the available approximate occurrences, *i.e.*, on the constraint relaxations that would lead to more occurrences, if the associated constraints were relaxed.

Removing a constraint suggested by the contradiction explanation does not necessarily lead to a new solvable CSP, but the constraints are relaxed recursively until a solvable CSP is obtained, or no constraints remain. The solutions of a new solvable CSP are approximate occurrences of the design pattern. Yet, while representing a design pattern that does not follow the theoretical definition proposed in [GHJV94], we only consider relaxing constraints that do not change the semantics of the original collaboration characterizing the pattern. Design pattern representations may be different from one system to another, but we do not adapt our representations to each one. Constraints suggested by the solver can be relaxed only if they verify the criteria expressed by the two main forms of approximate descriptions, as described in the next chapter.

In the case of the running example, the occurrences identified when solving the CSP are in the form:

```
<Sol.#>.var_createMemento   = <an entity>
<Sol.#>.var_newMemento      = <an entity>
<Sol.#>.var_setState        = <an entity>
<Sol.#>.var_setMemento      = <an entity>
<Sol.#>.var_getState        = <an entity>
<Sol.#>.var_caretaker       = <an entity>
<Sol.#>.var_originator      = <an entity>
<Sol.#>.var_memento         = <an entity>
```

and applying the solver to the set of constraints defining the Memento pattern, the CSP provides one and only one solution, without relaxations:

```
1.var_createMemento         = createMemento()
1.var_newMemento            = new Memento()
1.var_setState              = setState(String state)
1.var_setMemento            = setMemento()
1.var_getState              = getState()
1.var_caretaker             = Caretaker [14613018]
1.var_originator            = Originator [12386568]
1.var_memento               = Memento [17237886]
```

Since the running example has been implemented rigorously following the solution of the Memento pattern as proposed in [GHJV94], no approximate occurrences different from the complete occurrence were found by the solver.

5.3 Conclusion

In this chapter, we explained the use of constraint programming in our approach to identify complete and approximate occurrences of behavioral and creational design patterns in software systems. Since explanation-based constraint programming provides contradiction explanations when no more solution is found to a given problem, it is therefore possible to find approximate occurrences of a design pattern.

CHAPTER 6

EVALUATION

In this chapter, we first illustrate our approach using a case study on the identification of occurrences of the Visitor pattern in one particular scenario of JHotDraw. Then, we study the accuracy of our approach on several design patterns implemented in a set of Java systems in terms of precision and recall. Finally, we discuss threats to the validity of the evaluation of our approach.

6.1 JHotDraw Case Study

We already showed how to identify the Memento pattern in a simple toy system as a running example. Now, following the process in Figure 1.1, we show step-by-step identification of occurrences of the Visitor pattern in one scenario of JHotDraw v6.0b1. JHotDraw[4] is a Java GUI framework for the drawing of technical and structured graphics, letting a user create and manipulate figures. It has originally been developed by Erich Gamma—one of the co-authors of the book [GHJV94]—as a "design exercise". Its design strongly relies on some well-known design patterns. Among others, the application's user interface displays the documentation of the Visitor pattern when the copy and paste functionalities are activated by a user. The case study is based on these two functionalities.

6.1.1 Step 1: Description of the Visitor Pattern

Figure 6.1, inspired by [GHJV94], describes the scenario diagram of the Visitor pattern, based on our metamodel, as interactions between objects. A client that uses the Visitor pattern creates a `ConcreteVisitor` object and then goes through the object structure, visiting each element with the visitor. When an element is visited, it calls the `visitConcreteElement()` operation that corresponds to its

[4]http://www.jhotdraw.org/

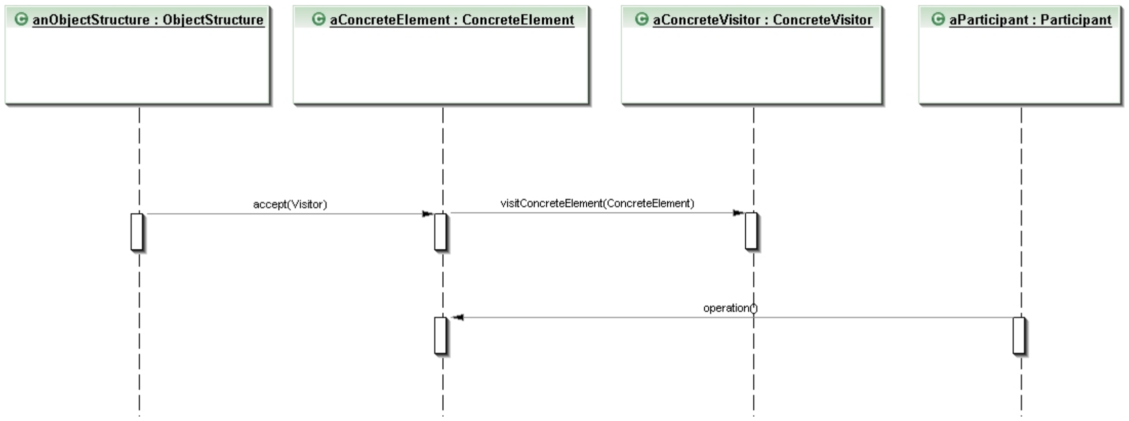

anObjectStructure : ObjectStructure
aConcreteElement : ConcreteElement
aConcreteVisitor : ConcreteVisitor
aParticipant : Participant
accept(Visitor)
visitConcreteElement(ConcreteElement)
operation()

Figure 6.1: Description of the Visitor pattern in terms of collaboration

class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary. In other words, the Visitor pattern is a way of separating an algorithm from an object structure by using double dispatch, giving the possibility for software developers to add new operations to existing object structures without modifying those structures. A typical example where the Visitor pattern is implemented is the case of a compiler that represents programs as abstract syntax trees for static analysis like checking that all variables are defined. It will also need to generate code. So it might define operations for type-checking, code optimization, flow analysis, checking for variables being assigned values before they are used, and so on.

6.1.2 Step 2: Reverse Engineering of Scenario Diagram

As described in Figure 1.1, Step 2i, JHotDraw bytecode is the output from a Java compiler. Once all Java class files are instrumented (Figure 1.1, Step 2ii), a chosen scenario (Figure 1.1, Step 2iii) of JHotDraw is executed (Figure 1.1, Step 2iv). We chose to test our identification approach on a scenario that is commonly executed by any user of JHotDraw, and which exercises the Visitor pattern, *i.e.*, *Cut and paste a figure in a document*:

```
1.  Create a new document on which figures can be drawn;
2.  Select the ‘Draw Rectangle’ tool from the menu;
3.  Draw a rectangle figure in the newly created document;
4.  Select the rectangle figure drawn at step 3;
5.  Select the ‘Cut’ command from the menu;
6.  Select the ‘Paste’ command from the menu.
```

6.1.3 Step 3: Constraint Satisfaction Problem

Following Figure 1.1, Step 3, we translate the description of the Visitor pattern into an eCSP.

Each instance of `Classifier` or `Message` in the scenario diagram instance of our metamodel in Figure 6.1, the source system, is associated with a variable in the eCSP bearing a similar name, *i.e.*, `var_anObjStruct`, `var_aConcreteElement`, `var_aConcreteVisitor`, `var_aParticipant`, `var_accept`, `var_visitConcreteElement`, and `var_operation`. The respective domain of variables `var_anObjStruct`, `var_aConcreteElement`, `var_aConcreteVisitor`, and `var_aParticipant` corresponds to the instances of `Classifier` appearing in the scenario diagram of the target system, where as the respective domain of variables `var_accept`, `var_visitConcreteElement`, and `var_operation` is the instances of `Operation` of the same scenario diagram.

The set of constraints among the entities is described below:

```
 1. notEqual(var_anObjStruct, var_aConcreteElement)
 2. notEqual(var_anObjStruct, var_aConcreteVisitor)
 3. notEqual(var_aConcreteVisitor, var_aConcreteElement)
 4. notEqual(var_aConcreteElement, var_aParticipant)
 5. follows(var_accept, var_visitConcreteElement)
 6. follows(var_visitConcreteElement, var_operation)
 7. follows(var_accept, var_operation)
 8. caller(var_anObjStruct, v_accept)
 9. callee(var_accept, v_aConcreteElement)
10. caller(var_aConcreteElement, var_visitConcreteElement)
11. callee(var_visitConcreteElement, var_aConcreteVisitor)
12. caller(var_aParticipant, var_operation)
13. callee(var_operation, var_aConcreteElement)
14. parameterCalleeSameType(var_accept, var_visitConcreteElement)
15. parameterCalleeSameType(var_visitConcreteElement, v_accept)
16. isContainedIn(var_operation, var_visitConcreteElement)
```

When solving the eCSP, we obtain three occurrences of the Visitor pattern. According to the documentation of JHotDraw, two are complete occurrences, whereas the other is an approximate occurrence.

Solution 1 is:

```
1.var_accept                 = visit(FigureVisitor visitor)
1.var_visitConcreteElement   = visitFigure(Figure hostFigure)
1.var_operation              = removeFromContainer(FigureChangeListener c)
1.var_objectStructure        = CutCommand [11197591]
1.var_concreteElement        = AnimationDecorator [24934792]
1.var_concreteVisitor        = DeleteFromDrawingVisitor [12741398]
1.var_aParticipant           = BouncingDrawing [6626965]
```

Solution 2 is:

```
2.var_accept                 = visit (FigureVisitor visitor)
2.var_visitConcreteElement   = visitFigure (Figure hostFigure)
2.var_operation              = setZValue (int z)
2.var_objectStructure        = ZoomDrawingView [5819561]
2.var_concreteElement        = AnimationDecorator [12839271]
2.var_concreteVisitor        = InsertIntoDrawingVisitor [2554341]
2.var_aParticipant           = BouncingDrawing [6626965]
```

Solution 3 is:

```
3.var_accept                 = visit (FigureVisitor visitor)
3.var_visitConcreteElement   = visitFigure (Figure hostFigure)
3.var_operation              = addToContainer (FigureChangeListener c)
3.var_objectStructure        = ZoomDrawingView [5819561]
3.var_concreteElement        = AnimationDecorator [12839271]
3.var_concreteVisitor        = InsertIntoDrawingVisitor [2554341]
3.var_aParticipant           = BouncingDrawing [6626965]
```

The value of the variables provided in Solution 1 and Solution 3 corresponds to the participants and messages involved in a complete occurrence of the Visitor pattern, respectively when the functionalities *cut a figure* and *paste a figure* are performed by the user of JHotDraw. In contrast, the value of variable `var_operation` in Solution 2, `public void setZValue(int)`, is involved in the sequence of messages corresponding to an approximate description of the Visitor pattern. A manual inspection of the source code revealed that `addToContainer (FigureChangeListener c)` and `public void setZValue(int)` are both triggered while visiting a figure that is to be added in a document.

6.2 Accuracy on Several Systems

We now evaluate our approach on a set of software systems using the measures of precision and recall typically used in the domain of information retrieval [FBY92]. Precision assesses the proportion of true occurrences of a design pattern among all the occurrences identified by our analysis given a scenario of a system, while recall assesses the proportion of true occurrences of a design pattern identified by our analysis among all the ones really existing in the given scenario:

$$precision = \frac{|\{existing\ occurrences\} \bigcap \{identified\ occurrences\}|}{|\{identified\ occurrences\}|}$$

$$recall = \begin{cases} \frac{|\{existing\ occurrences\} \bigcap \{identified\ occurrences\}|}{|\{existing\ occurrences\}|}, & \text{if } |\{existing\ occurrences\}| \neq 0 \\ 100\%, & \text{otherwise} \end{cases}$$

Table 6.1 reports the precision and recall of our approach for five systems evaluated on three design patterns. The tests were made on an AMD Athlon 64bit x 2 Dual machine at 2.41GHz. The set of software systems written in Java includes: DRESDEN OCL v1.1, JHOTDRAW v6.0b1, JREFACTORY v2.6.24, PMD v1.8., and QUICKUML 2001. DRESDEN OCL[5] is a modular OCL (Object Constraint Language) toolkit that parses and type-checks OCL constraints and instruments Java code for runtime verification. It is also integrated into various CASE-tools and provides an SQL-code generator. JREFACTORY[6] is a refactoring tool for the Java programming language that includes a pretty printer, a UML class diagram viewer, a coding standards checker, and computes system metrics. PMD[7] is a Java source code analyzer that finds unused variables, empty catch blocks, unnecessary object creation, and so forth. Finally, QUICKUML[8] is a class-diagram graphic editor that

[5]http://dresden-ocl.sourceforge.net/
[6]http://jrefactory.sourceforge.net/
[7]http://pmd.sourceforge.net/
[8]http://www.excelsoftware.com/quickumlwin.html

SYSTEM / SCENARIO	DESIGN PATTERN	BUILDER	COMMAND	VISITOR
Dresden OCL	*Transform OCL into SQL*	5/5 = 100% (5/5 = 100%) 100%	-	-
JHotDraw	*Cut and paste a rectangle*	-	100%	3/3 = 100% (2/3 ≅ 66,7%) 100%
	Align figures	-	1/2 = 50% (1/3 ≅ 33,3%) 100%	-
	Bring figures to front	-	1/1 = 100% (1/4 = 25%) 100%	-
	Send figures to back	-	1/1 = 100% (1/4 = 25%) 100%	-
	Group figures	-	1/1 = 100% (1/3 ≅ 33,3%) 100%	-
JRefactory	*Calculate a set of metrics of a class*	-	-	11/13 ≅ 84,6% (11/13 ≅ 84,6%) 100%
PMD	*Find variables with short names*	-	-	1/1 = 100% (1/1 = 100%) 100%
QuickUML	*Resize a diagram*	-	-	2/2 = 100% (2/2 = 100%) 100%
	Enable toggle refresh from the menu	100%	1/2 = 50% (1/3 ≅ 33,3%) 100%	-
	Build a class from UML	1/1 = 100% (1/2 = 50%) 100%	100%	-
AVERAGE PRECISION		100% (75%)	80% (30%)	96,2% (87,8%)

Table 6.1: Precision and recall calculated on particular scenarios of software systems for which the uses of design patterns are known. (For each row, the first line shows two different ways of measuring precision: the first one takes into account approximate descriptions of the design pattern, while the precision in parenthesis considers only complete occurrences of the design pattern. The second line is the recall.)

tightly integrates a core set of UML models. We chose these systems because they are open-source, and thus one can reproduce our experimentations without being limited by proprietary source code.

Typically, in our experiments, the size of an execution trace varies from 450 Ko (2 310 messages executed) to 5.5 Mo (19 620 messages executed), depending on the size of the system, and the complexity of the executed scenario. The average time of calculation of solutions, from the moment that an execution trace is transformed into an instance of the scenario diagram metamodel, to the computation of design pattern occurrences, varies from a few minutes to seven days. The larger the size of an execution trace is, the more computation time the backtracking mechanism behind the constraint solver requires to find occurrences matching a design pattern.

During our experimentations, while studying the accuracy of the new solutions found by relaxing constraints of the original problem suggested by the solver, *i.e.*, the approximate occurrences of a design pattern, we categorized into two types the possible approximate descriptions that still kept the principle of the design pattern after constraint relaxation. First, we observe that it is a common practice to add intermediate participants and messages to its original collaboration while implementing a design pattern. Second, some collaborating messages are often represented by more than one operation during implementation. For example, regarding the representation of the `Command` pattern in Figure 3.4 on page 26, the `ConcreteCommand` object invokes operations modelled by message `action()` on its receiver to carry out a request. Clearly, during implementation of a real-world system, it is uncommon to see only *one* message carrying out the request of the invoker. Instead, a set of messages is called to complete the request.

Therefore, results of precision are reported in Table 6.1 in two different ways: the first one takes into account approximate descriptions of the design pattern, while the precision in parenthesis considers only complete occurrences of the design pattern.

In average, we observe that precision and recall are both very high for each pair of *scenario/design pattern* when the evaluation concerns approximate descriptions of the design pattern, as opposed to poor results in certain cases (33%, 25%) for complete occurrences of the design pattern. After manual inspection of the pattern occurrences suggested by our tool versus those documented, we can explain these contrasting precisions as a result of non-exact implementations of the documented patterns by the software. By relaxing certain constraints as suggested by the JChoco solver following the aforementioned criteria, precision drastically increases.

We also applied the identification approach on a subset of *scenario/design pattern* of Table 6.1, for which the corresponding design pattern is known to not be implemented. Results show that no occurrence of the searched design patterns was identified for the specific scenarios. Therefore, for these evaluated scenarios, no false positives were found by our approach. Following the specified definition of recall, results appear as 100% in Table 6.1, without any measures of recall and precision in parenthesis.

In conclusion, our approach works for several systems and various design patterns, without providing many false positive occurrences and missing many true positive occurrences. Although the executed scenarios for each evaluated system are not an exhaustive list of all the possible scenarios, the current results are good predictors of the accuracy of the proposed approach for the identification of behavioral and creational design patterns using dynamic analysis. In future work, we will assess the precision and recall on other design patterns and software systems.

6.3 Threats to Validity

Despite its interesting results, our approach must be considered in the light of threats to validity. In this section, we explain how valid our results are, following the classification schemes of Campbell and Stanley [CS63] for two types of threats to the validity of an experiment: threats to internal validity and threats to external validity.

6.3.1 Internal Validity

If a relationship is observed between the treatment and the outcome, we must make sure that it is a causal relationship, and that it is not a result of a factor over which we have no control or which we have not measured. Threats to internal validity concern issues that may indicate a causal relationship, although there should be none.

In this thesis, these issues include the accuracy of the complete and approximate occurrences of a design pattern that our approach identifies: *are they occurrences that are as a matter of fact implemented in the evaluated system?* To counter this threat, we specifically evaluated systems that are known in the community to have very clear implementations of the design patterns in [GHJV94]. Among them, some systems even have explicit class and method names of the implemented patterns (yet, one should note: our identification approach is independent of the class and method names). Evaluating systems for which the presence of a design pattern is known is mandatory, because the measures of precision and recall are calculated with the number of existing occurrences of the searched design pattern. Since the set of evaluated systems are well documented and have already been the subjects of evaluation in previous design pattern identification approaches, they form a good benchmark for the evaluation, and eliminate a threat to the internal validity of our results.

6.3.2 External Validity

The external validity is concerned with generalization. If there is a causal relationship between the construct of the cause, and the effect, can the result of the study be generalized outside the scope of our study? Is there a relation between the treatment and the outcome? Threats to external validity concern the ability to generalize experiment results outside the experiment setting. In our experimentations, factors that impact on the external validity are how the systems and scenarios are chosen.

In our experiments, these issues include four aspects that are explained in the subsequent four sections.

6.3.2.1 Choice of the Systems

The systems on which we apply our identification approach are all in Java. This fact represents a threat to external validity: *can our identification approach be generalized for other programming languages apart from Java?* As previously discussed in Chapter 4, the target language influences only how an execution trace is obtained, but does not affect the scenario diagram, since our metamodel captures the dynamic semantic of design patterns, and can therefore accommodate any mainstream object-oriented programming language. Furthermore, the fundamental principles that we proposed in Chapter 4 to instrument the execution of a system can be generalized to most object-oriented programming languages.

6.3.2.2 Choice of the Scenarios

The choice of scenarios that are executed for each evaluated system is another threat to external validity: *are the executed scenarios representative enough for the generalization of the accuracy of our results?* While evaluating our approach, we chose a subset of scenarios to be executed. They were chosen in terms of their representativeness of the underlying software system. An ideal evaluation requires to trigger as many scenarios as possible, to evaluate the design pattern identification approach on all of the functionalities of a system. However, due to the limited time for the completion of this thesis, we only used a subset of the scenarios representing the most commonly used functionalities, whose description is generally available in the documentation. Future work includes the merging of several scenario diagrams to obtain one sequence diagram.

6.3.2.3 Performance

One of the key challenges of dynamic analysis is to cope with the large amount of data generated by system monitoring. As the size of the target system grows, execution traces also grow, and the computation time required to solve eCSPs increases.

To cope with high data volume , we chose to use start and end markers to specify respectively the start and end of the action primary to a particular scenario. For example, in the *Cut and paste a figure in a document* scenario described in Section 6.1.2, the two principal actions involved are actions *cut* and *paste*. We thus placed two markers in the execution trace of the corresponding scenario to specify the start and end of action *cut*, just before and after the user executes *cut* from the menu of JHotDraw. In the same manner, two markers are specified respectively for the start and end of action *paste*. Method and constructor executions that are positioned outside each pair of start and end markers can be omitted. Results after applying our identification approach, both on the original and the shorten execution trace, return identical occurrences of the Visitor pattern. The marker mechanism reduces the volume of dynamic data, but still needs some more refinement to assure that no occurrences of a design pattern are omitted because some method and constructor executions are eliminated from the original execution trace.

We also consider using an abstraction mechanism, such as summarizing an execution trace to extract only its main content. Such strategy removes from the trace implementation details like calls to utility methods, which have no obvious use for the identification of behavioral and creational design patterns. Some work has already been done in that direction, for example [HLBAL05, HLL06], and we plan to reuse such an approach to further decrease the size of execution traces, and thus improve performances.

6.3.2.4 Description of Design Patterns

As explained in Chapter 3, we describe design patterns in terms of collaboration among objects given in [GHJV94]. However, as classes participating in a design pattern need not be collaborating precisely according to the proposed collaboration, the design pattern description could be done in such a way that users could easily describe the collaboration among participants to characterize their own patterns of interest. Then, its translation into an eCSP can be automated.

6.4 Conclusion

In this chapter, we provided a case study showing in detail how to concretely perform the Visitor pattern identification over one scenario of JHotDraw following our approach. Then, we evaluated our approach on chosen scenarios taken from five different systems written in Java, by providing precision and recall results. Finally, we discussed threats to the validity of the evaluation.

In the next chapter, we will first provide a summary of the work achieved in this thesis, and then suggest future work.

CHAPTER 7

CONCLUSION

The problem of identifying design patterns in software systems has been the subject of several work in the past few years. In particular, authors think that the recovery of design patterns applied during the phases of conception and implementation can facilitate software comprehension, and thus, maintenance. Most of the work focused on the use of static analysis to recover the elements, structures, and relationships describing the system. Work done in this domain of research demonstrated the accuracy of their approach on structural pattern identification. However, results are not as interesting when authors chose to identify behavioral and creational patterns with static analysis: too many false positives were generated. The reason is that behavioral and creational patterns are typically characterized by the interactions between the participants of the design pattern, and consequently, structural data alone do not provide sufficient significant information. Therefore, we proposed a 3-step approach to identify behavioral and creational design patterns in source code using dynamic analysis.

We presented other approaches that combined dynamic data with static analysis for the recovery of structural patterns. Inspired by the work of Briand *et al.* [BLL06], we proposed a partial metamodel composed of meta-entities partially modelling UML sequence diagram elements. Among others, messages of type operation calls, creation calls, and destruction calls, as well as combined fragments of type loops and alternatives are meta-entities used to model interactions between participants of a scenario diagram when a scenario of a system is executed. Then, for each behavioral and creational design pattern in [GHJV94] for which a description of the interactions between participants is available, *i.e.*, the Builder, Command, Memento, Observer, and Visitor patterns, we showed how to describe it with its dynamic properties, by instantiating the scenario diagram metamodel. To reverse engineer scenario diagrams from a given system over which we want to perform pattern identification, we first proceeded with the instrumentation of the system to obtain execution traces recording the messages sent for each executed scenario. In particular, we emphasized on the instrumentation strategy to trace constructor and method executions, as well as on control flow and repetition of messages. In turn, these dynamic data gathered in one execution trace are transformed into an instance of the scenario diagram metamodel, the same one used to describe a design pattern. However, the execution trace is limited in tracing only one particular scenario of the system. Finally, we performed the concrete pattern identification using explanation-based constraint programming by identifying, in the scenario diagram of an executed scenario of a system, objects and messages in conformity with the set of constraints derived from the scenario diagram of the searched design pattern. We evaluated our approach on Dresden OCL Toolkit, JHotDraw, JRefactory, PMD, and QuickUML with the Builder, Command, and Visitor patterns to show its precision and recall. We showed that using dynamic analysis, results are promising for the identification of behavioral and creational design patterns.

Therefore, in this thesis, we showed that it is possible to identify pure behavioral and creational design patterns using dynamic analysis only. It will be interesting to do more research studying the synergy between static and dynamic approaches, to see how they complement each other.

Future Work

Future work includes evaluating our approach on more behavioral and creational design patterns and software systems; merging scenario diagrams to obtain sequence diagrams; using abstraction and summarization mechanisms that can reduce the size of an execution trace without the loss of data relevant to the identification of design patterns, to address performance and scalability issues; adding new constraints and improving the eCSP of the design patterns to obtain higher precision without impacting recall; combining our approach with previous structural approaches.

BIBLIOGRAPHY

[AAG01] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Pim van den Broek, Pavel Hruby, Motoshi Saeki, Gerson Sunyé, and Bedir Tekinerdogan, editors, *Proceedings of the* 1st *ECOOP Workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, October 2001. TR-CTIT-01-35.

[ACdPF01] Giuliano Antoniol, Gerardo Casazza, Massimiliano di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59:181–196, November 2001.

[Apa06] Apache Jakarta Project. *Byte Code Engineering Library*, June 2006.

[Bes91] Christian Bessière. Arc-consistency in dynamic constraint satisfaction problems. In Thomas L. Dean and Kathleen McKeown, editors, *Proceedings of the* 9th *National Conference on Artificial Intelligence*, pages 221–226. AAAI Press / The MIT Press, July 1991.

[BLL06] Lionel Briand, Yvan Labiche, and Johanne Leduc. Towards the reverse engineering of UML sequence diagrams for distributed Java software. *Transactions on Software Engineering*, 32(9):642–663, September 2006.

[BLM03] Lionel Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. *Proceedings of the* 10th *Working Conference on Reverse Engineering*, pages 57–66, November 2003.

[BM96] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In Dan Weld and Bill Clancey, editors, *Proceedings of the*

13th *National Conference on Artificial Intelligence*, pages 298–304. AAAI Press / The MIT Press, August 1996.

[CS63] Donald T. Campbell and Julian C. Stanley. *Experimental and Quasi-Experimental Designs for Research.* Rand McNally College Publishing, 1963.

[DHKV93] Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides. An architecture for visualizing the behavior of object-oriented systems. *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1993.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[FBY92] William B. Frakes and Ricardo A. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, 1992.

[GAA01] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *Proceedings of the 39th Conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.

[GAA04] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1st edition, 1994.

[Gin93] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[GJ01] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *Proceedings of the* 1st *IJCAI Workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[GJP00] Christelle Guéret, Narendra Jussien, and Christian Prins. Using intelligent backtracking to improve branch and bound methods: An application to open-shop problems. *European Journal of Operational Research*, 127(2):344–354, December 2000.

[GZ05] Yann-Gaël Guéhéneuc and Tewfik Ziadi. Automated reverse-engineering of uml v2.0 dynamic models. In Serge Demeyer, Stéphane Ducasse, Kim Mens, and Roel Wuyts, editors, *Proceedings of the* 6th *ECOOP Workshop on Object-Oriented Reengineering*. Springer-Verlag, July 2005.

[HHL02] Dirk Heuzeroth, Thomas Holl, and Welf Löwe. Combining static and dynamic analyses to detect interaction patterns. In Hartmut Ehrig, Bernd J. Krämer, and Atila Ertas, editors, *Proceedings of the* 6th *International Conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.

[HLBAL05] Abdelwahab Hamou-Lhadj, Edna Braun, Daniel Amyot, and Timothy Lethbridge. Recovering behavioral design models from execution traces. In *CSMR '05: Proceedings of the* 9th *European Conference on*

Software Maintenance and Reengineering, pages 112–121, Washington, DC, USA, 2005. IEEE Computer Society.

[HLL06] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

[HLM03] Dirk Heuzeroth, Welf Löwe, and Stefan Mandel. Generating design pattern detectors from pattern specifications. In *18th IEEE International Conference on Automated Software Engineering (ASE) 2003*, pages 245–248. IEEE, 2003.

[JB97] Narendra Jussien and Patrice Boizumault. Best-first search for property maintenance in reactive constraints systems. In *International Logic Programming Symposium*, pages 339–353, Port Jefferson, N.Y., USA, October 1997. MIT Press.

[JB00] Narendra Jussien and Vincent Barichard. The PaLM system: Explanation-based constraint programming. In Nicolas Beldiceanu, Warwick Harvey, Martin Henz, François Laburthe, Eric Monfroy, Tobias Müller, Laurent Perron, and Christian Schulte, editors, *Proceedings of TRICS: Techniques foR Implementing Constraint Programming Systems*, pages 118–133. School of Computing, National University of Singapore, Singapore, September 2000. TRA9/00.

[JDB00] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In Rina Dechter, editor, *Proceedings of the* 6^{th} *Conference on Principles and Practice of Constraint Programming*, pages 249–261. Springer-Verlag, September 2000.

[Jik08] Jikes RVM. *Jikes RVM*, 2008.

[JL00] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In Henry A. Kautz and Bruce Porter, editors, *Proceedings of the* 17^{th} *National Conference on Artificial Intelligence*, pages 169–174. AAAI Press / The MIT Press, July–August 2000.

[JSB97] Dean Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. *Proceedings of the 1997 International Conference on Software Engineering*, pages 360–370, May 1997.

[Jun01] Ulrich Junker. QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. Technical report, Ilog SA, 2001.

[Jus01] Narendra Jussien. e-Constraints: Explanation-based constraint programming. In Barry O'Sullivan and Eugene Freuder, editors, 1^{st} *CP Workshop on User-Interaction in Constraint Satisfaction*, December 2001.

[KP96] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In Linda M. Wills and Ira Baxter, editors, *Proceedings of the* 3^{rd} *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In David Garlan and Jeff Kramer, editors, *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[NG07] Janice Ka-Yee Ng and Yann-Gaël Guéhéneuc. Identification of behavioral and creational design patterns through dynamic analysis.

In Andy Zaidman, Abdelwahab Hamou-Lhadj, and Orla Greevy, editors, *Proceedings of the* 3^{rd} *International Workshop on Program Comprehension through Dynamic Analysis*, pages 34–42. Delft University of Technology, October 2007. TUD-SERG-2007-022.

[Nie02] Jörg Niere. Fuzzy logic based interactive recovery of software design. Presented at the ICSE Doctoral Symposium, May 2002.

[Obj04] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004.

[OS02] Rainer Oechsle and Thomas Schmitt. Javavis: Automatic program visualization with object and sequence diagrams using the java debug interface (jdi). In *Revised Lectures on Software Visualization, International Seminar*, pages 176–190, London, UK, 2002. Springer-Verlag.

[PJM$^+$02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, 2002. Springer-Verlag.

[PKV94] Wim De Pauw, Doug Kimelman, and John M. Vlissides. Modeling object-oriented program execution. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, volume 821, pages 163–182. Springer-Verlag, July 1994.

[PL06] Niklas Pettersson and Welf Lowe. Efficient and accurate software pattern detection. In *Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 317–326, Washington, DC, USA, 2006. IEEE Computer Society.

[QYW97] Alex Quilici, Quing Yang, and Steven Woods. Applying plan recognition algorithms to program understanding. *Journal of Automated Software Engineering*, 5(3):347–372, July 1997.

[RC05] Atanas Rountev and Beth Harkness Connell. Object naming analysis for reverse-engineered sequence diagrams. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 254–263, New York, NY, USA, 2005. ACM.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of* 7th *International Conference on Software Maintenance*, pages 13–22. IEEE Computer Society Press, August 1999.

[RVR05] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. *Proceedings of the* 6th *Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, September 2005.

[SAEHES05] Doaa M. Shawky, Salwa K. Abd-El-Hafiz, and Abdel-Latif El-Sedeek. A dynamic approach for the identification of object-oriented design patterns. *Proceedings of the* 2nd *International Conference on Software Engineering*, pages 138–143, February 2005.

[SKM01] Tarja Systa, Kai Koskimies, and Hausi Muller. Shimba–an environment for reverse engineering java software systems. *Software Practice and Experience*, 31(4):371–394, 2001.

[SV94] Thomas Schiex and Gérard Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, February 1994.

[SvG98] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of Java software. In Bill Scherlis, editor, *Proceedings of* 5th *International Symposium on Foundations of Software Engineering*, pages 10–16. ACM Press, November 1998.

[TCSH06] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11), November 2006.

[TP03] Paolo Tonella and Alessandra Potrich. Reverse engineering of the interaction diagrams from C++ code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, pages 159–168, Washington, DC, USA, 2003. IEEE Computer Society.

[WMFB+98] Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. *SIGPLAN Not.*, 33(10):271–283, 1998.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *Proceedings of the* 26th *Conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.

Appendix I

Design Patterns Glossary

Creational Patterns

Abstract Factory Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Builder Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Factory Method Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Prototype Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

Singleton Ensure a class only has one instance, and provide a global point of access to it.

Structural Patterns

Adapter Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Bridge Decouple an abstraction from its implementation so that the two can vary independently.

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

Decorator Attach additional responsibilities to an object dynamically. Decorator provides a flexible alternative to subclassing for extending functionality.

Facade Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Flyweight Use sharing to support large numbers of fine-grained objects efficiently.

Proxy Provide a surrogate or placeholder for another object to control access to it.

Behavioral Patterns

Chain of Responsibility Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Command Encapsulate a request as an object, thereby parameterizing clients with different requests, queue or log requests, and support undoable operations.

Interpreter Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Iterator Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Mediator Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and their interaction can be varied independently.

Memento Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

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

State Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

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

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

Visitor Represent an operation to be performed on the elements of an object structure. Visitor allows defining a new operation without changing the classes of the elements on which it operates.