

# Identification of Behavioral and Creational Design Patterns through Dynamic Analysis

Janice Ka-Yee Ng and Yann-Gaël Guéhéneuc

PTIDEJ Team – GEODES

Département d’informatique et de recherche opérationnelle

Université de Montréal – CP 6128 succ. Centre Ville

Montréal, Québec, H3C 3J7 – Canada

ngjanice@iro.umontreal.ca and guehene@iro.umontreal.ca

## Abstract

*Design patterns are considered to be a simple and elegant way to solve problems in object-oriented software systems, because their application leads to a well-structured object-oriented design, and hence, are considered to ease software comprehension and maintenance. However, due to the complexity of large object-oriented software systems nowadays, it is impossible to recover manually the design patterns applied during the design and implementation of a system, which, in turn, impedes its comprehension. In the past few years, the structure and organization among classes were the predominant means of identifying design patterns in object-oriented software systems. In this paper, we show how to describe behavioral and creational design patterns as collaborations among objects and how these representations allow the identification of behavioral and creational design patterns using dynamic analysis and constraint programming.*

## 1 Introduction

In the past, several approaches have been proposed to detect design patterns in source code using static analysis. The fundamental idea of these approaches consists in analyzing the class structure of a system to identify classes whose structure resembles the most the structure of a design pattern. The dynamic aspect of the system has almost been completely ignored, but it should not be because, on the one hand, behavioral and creational design patterns can hardly be described by their structure, and on the other hand, the dynamic aspect provides data to complement data related to the architecture and design of software systems.

In this paper, we propose a 3-step approach (as illustrated in Figure 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 design patterns in terms of UML sequence diagrams (really scenario diagrams as explained in Section 2). Second, using dynamic analysis, we reverse engineer a dynamic model—such as UML sequence diagrams—of any given object-oriented software system written in Java. Finally, we perform the **Visitor** pattern identification on one particular scenario of JHOTDRAW. To this end, we translate the problem of design patterns identification in terms of a constraint satisfaction problem (a.k.a. CSP).

This paper is structured as follow: In Section 2, we provide a metamodel to capture the interactions between objects at runtime. Then, a description of behavioral and creational design patterns in terms of the constructs of this metamodel is provided in Section 3. In Section 4, we describe our technique to reverse engineer scenario diagram of object-oriented software systems using dynamic analysis. Section 5 elaborates on the technique used to identify behavioral and creational design patterns. We then report the results of one case study in Section 6. Challenges and limitations of the approach are discussed in Section 7. Related work is provided in Section 8. Finally, Section 9 concludes and presents future work.

## 2 Scenario Diagram Metamodel

In the context of design patterns identification, the reverse engineered UML sequence diagrams are obtained from the execution of some particular use cases. Therefore, these diagrams are referred to as *scenario diagram* [3], as they are only partial UML sequence di-

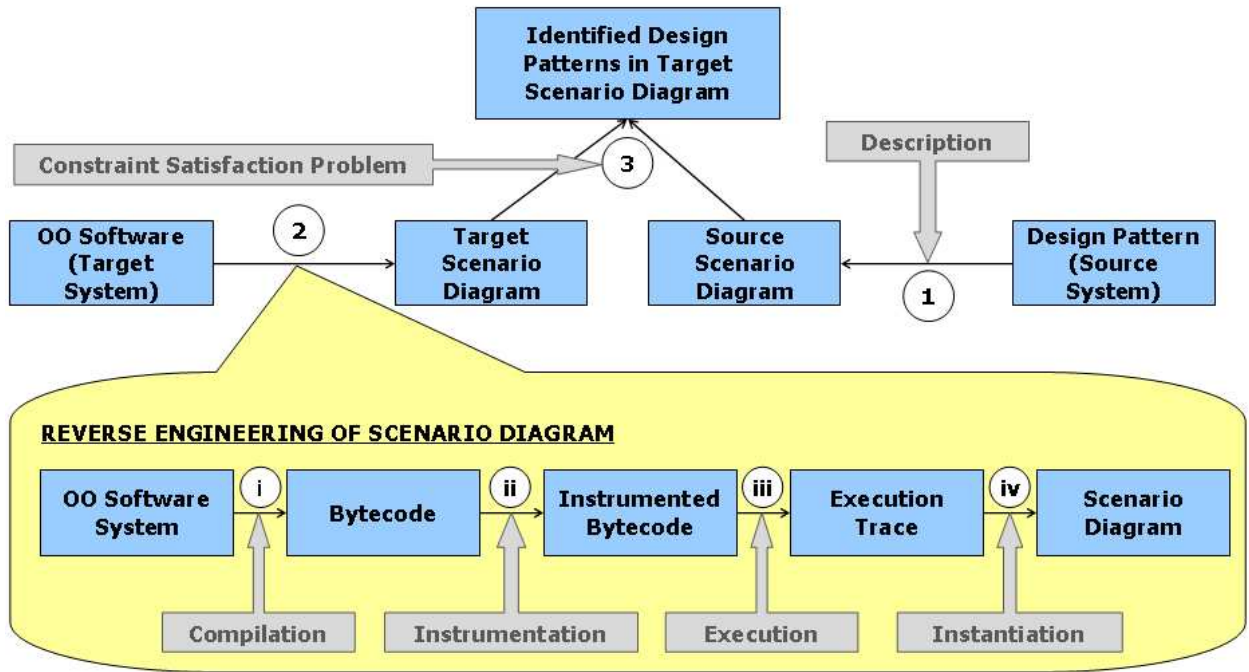


Figure 1. A 3-step approach for the identification of design patterns through dynamic analysis.

agrams describing one specific scenario corresponding to a use case instead of all possible alternatives for the exercised use case.

Following [3] and [12], we implement a metamodel of scenario diagrams to express the data we need to describe the behavior of design patterns and software systems. Figure 2 shows our scenario diagram metamodel. 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` that can be specialized into an `Instance` or a `Class`, the latter case is applicable if the message in relation to the caller or callee is a class method. If any, messages are composed of arguments, class `Argument`, of different types: either primitive types or object types. The return value of messages is class `ReturnValue`.

Class `CombinedFragment` is inspired by a previous notation [12] to group sets of messages to show conditional flows in sequence diagrams. Although [12] provides eleven interactions types of combined fragments,

only the combined fragments loops and alternatives are necessary to behavioral and creational design patterns identification. In this context, combined fragments can be specialized into two types: either loops, class `Loop`, to illustrate repetitions of messages, or alternatives, class `Alt`, to designate mutually exclusive choices between sequence of messages. To model the case where a loop or an alternative is nested into another loop or alternative, we introduce composition links: composition `operand` between classes `Loop` and `Component`, and composition `operands` 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’.

### 3 Description of Design Patterns

In [5], each design pattern is provided with their own description in terms of collaborations between participants. In particular, for some specific patterns, the authors have chosen to use diagrams similar to scenario diagrams to show sequences of messages between objects, i.e., the order in which messages between participants of design patterns are executed. For instance, Figure 3 is a scenario diagram that illustrates how the participants of the `Memento` pattern collaborate.

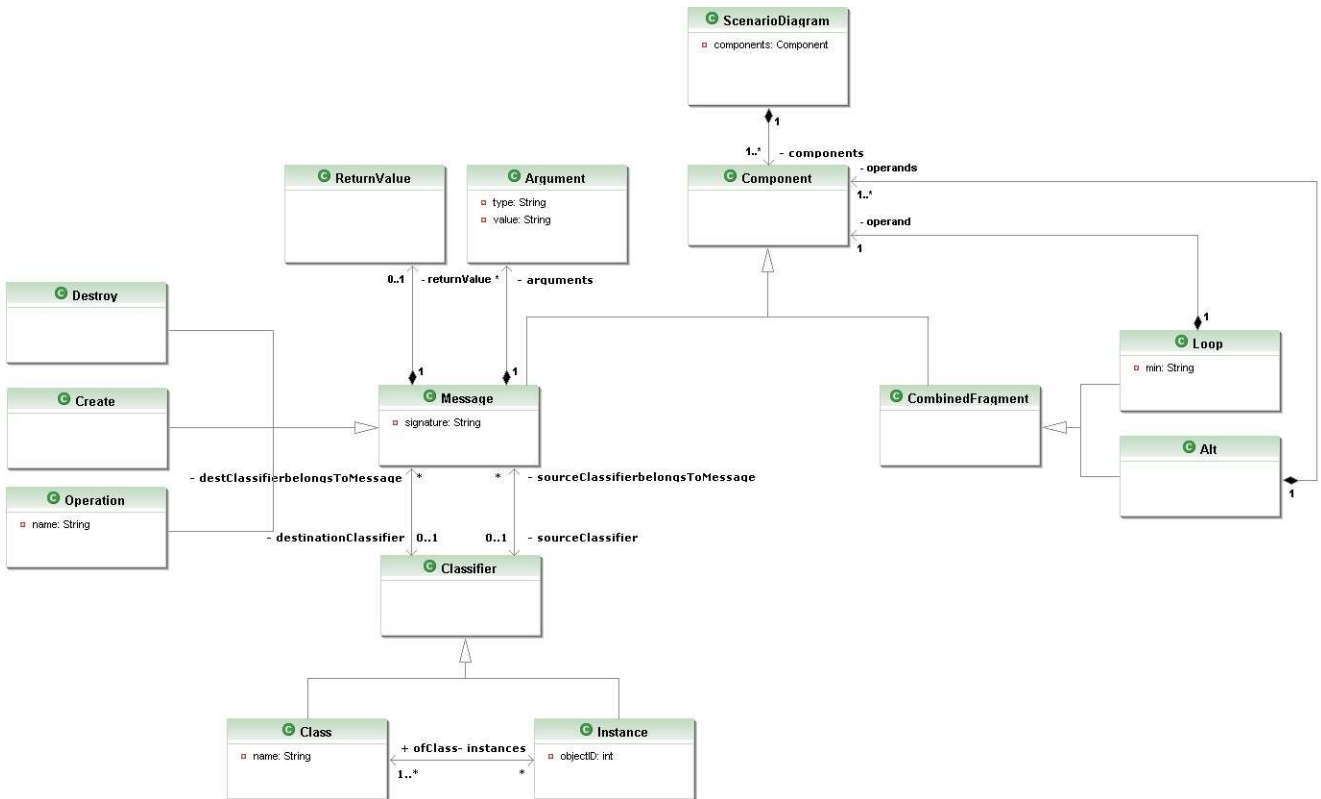


Figure 2. Scenario diagram metamodel.

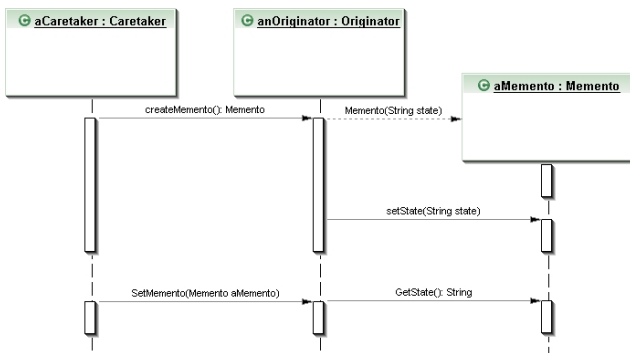


Figure 3. Memento pattern scenario diagram.

In our approach, as illustrated in Figure 1 Step 1, we describe behavioral and creational design patterns by transforming the graphical description of collaborations in [5] into an instance of the scenario diagram metamodel. For each design pattern for which a graphical description is available, we describe its participants and its sequence of messages in terms of objects of the scenario diagram metamodel. For instance, for each message involved in the sequence of messages

in Figure 3, we instantiate an object Operation that is added to the ordered list components of an object ScenarioDiagram representing the Memento pattern. Given message `setMemento(Memento aMemento)` takes ‘aMemento’ as argument, we instantiate an object Argument whose attribute `type` is Instance. This object Argument is added to the ordered list arguments of message `setMemento(Memento aMemento)`. The participants collaborating in the pattern, for instance aCaretaker, anOriginator, and aMemento, are instantiated as objects Instance, and are set to be the sourceClassifier or destinationClassifier of the corresponding message. For example, the sourceClassifier of `createMemento()` and `setMemento(Memento aMemento)` is aCaretaker, while anOriginator is their destinationClassifier.

## 4 Reverse Engineering of Scenario Diagram

In the literature, many approaches have been proposed to reverse engineer dynamic models of object-oriented software systems. Based on [3], our approach for the scenario diagram reverse engineering consists

in 4 steps (as illustrated in Figure 1 Steps i, ii, iii, and iv). First, we compile the source files of an object-oriented software system to obtain their corresponding class files. Second, we instrument the class files using bytecode instrumentation. Third, we execute the instrumented system following some scenario to produce an execution trace. Finally, we instantiate the scenario diagram that corresponds to the execution trace.

In this section, we describe briefly the mechanism used to produce an execution trace containing dynamic data of an object-oriented software systems in Java (Step ii), and to instantiate a scenario diagram from the execution trace (Step iv).

**Instrumentation.** In terms of design patterns identification, the type and amount of dynamic data to retrieve are relative to the description of design patterns. In this context, we focus primarily on the control flow data, that is, the sequence of messages actually executed during runtime. We have chosen to instrument Java bytecode with BCEL—the Byte Code Engineering Library [2]. BCEL is a Java library that gives users the possibility to create, analyze, and manipulate easily Java class files.

We need to trace the execution of methods and constructors to instantiate class `Message` in a scenario diagram. To this end, we introduce bytecode instructions to produce dynamic data before and after the execution of the methods and constructors. We indicate in the execution trace when methods and constructors start and end executing in relation to other events. Figure 4 shows an example of execution trace of a toy system implemented in Java.

**Instantiation of Scenario Diagram.** To obtain the scenario diagram corresponding to an execution trace, the latter is processed. For each execution trace statement such as `operation start`, `constructor start`, or `destructor start`, a message of type `Operation`, `Create`, or `Destroy` is respectively instantiated, while an object `CombinedFragment` of type `Loop` or `Alt` is instantiated for each execution trace statement `loop start` or `alt start`. In both cases, the component corresponding to the line 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 statement 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 determined and instantiated. The set `arguments` of a message is determined by processing the data positioned between the brackets of the corresponding execution statement. Figure 5 is a textual description of the scenario diagram corresponding to the execution trace in Figure 4.

## 5 Identification of Design Patterns

Using the reverse engineering technique described in the previous section, we instantiate two scenario diagrams. One instance models the sequence of messages of a design pattern, i.e., a *source system*, and the other instance models the sequences of messages of a given source code, i.e., a *target system*. The approach we propose to identify behavioral and creational design patterns in object-oriented software systems consists in identifying the scenario diagrams of a source system in the scenario diagram of a target system.

As illustrated in Figure 1 Step 3, we translate the problem of design patterns identification in terms of a constraint satisfaction problem (CSP, as in previous work [7]). We define the problem of detecting a design pattern in terms of its variables, the constraints among them, and their domains. This CSP represents the problem that the explanation-based constraint solver JCHOCO [11] solves to identify in the target system, sequence of messages that is identical or similar to the one defined by the source system.

**Variables.** The set of variables corresponds to the entities `Classifier` and `Message` modelling the scenario diagram of a design pattern (the source system).

**Constraints.** The set of constraints among the variables corresponds to the relationships among the entities of the scenario diagram defined by a design pattern. We use binary constraints, of the form `constraint(variable1, variable2)`, to express the relationships between `variable1` and `variable2`.

**Domain.** The domain of the variables corresponds to the entities modelling the target system. It consists of a set of integers, each corresponding to an entity in the scenario diagram of a target system.

For a given set of constraints, the constraint solver JCHOCO solves the CSP by removing from the domains values that do not satisfy the relationships between `variable1` and `variable2`. If the constraint solver JCHOCO provides no solution for a CSP, then the corresponding design pattern is considered as not implemented in the target system.

```

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()o 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. Example of execution trace of a toy system implementing the Memento Pattern.**

```

1 <OPERATION> public static void main (String[] args) <CALLEE> ModelMementoTest <CALLER> inexistant
2 <CREATE> public void <init>() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
3 <CREATE> public void <init>() <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
4 <OPERATION> public void callCreateMemento() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
5 <OPERATION> public Memento createMemento() <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
6 <CREATE> public void <init>() <CALLEE> Memento 17237886 <CALLER> Originator 12386568
7 <OPERATION> public void setState(String state) <CALLEE> Memento 17237886 <CALLER> Originator12386568
8 <OPERATION> public void undoOperation() <CALLEE> Caretaker 14613018 <CALLER> ModelMementoTest
9 <OPERATION> public void setMemento(Memento m) <CALLEE> Originator 12386568 <CALLER> Caretaker 14613018
10 <OPERATION> public String getState() <CALLEE> Memento 17237886 <CALLER> Originator 12386568

```

**Figure 5. Textual representation of the scenario diagram of Figure 4.**

The constraint caller (classifier1, message2) (respectively callee) defines the relationship ‘*classifier1 is the sourceClassifier of message2*’ (respectively destinationClassifier) between classifier1 and message2. 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 possible value taken by message2, there must be a corresponding value taken by classifier1 so that any Classifier in the domain of classifier1 is the sourceClassifier of a Message in the domain of message2. Conversely, for each possible value taken by classifier1, there must be a corresponding value taken by message2 so that the sourceClassifier of any Message in the domain of message2 is a Classifier in the domain of classifier1. Each value of classifier1 and message2 failing to comply to this relationship are removed from the corresponding domains.

The constraint follows(message1, message2) defines the relationship ‘*message2 is executed after message1*’. The domain of variable message1 and

message2 correspond 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 any Message in the domain of message2 is called after a Message in the domain of message1. Conversely, for each possible value taken by message1, there must be a corresponding value taken by message2 so that any Message in the domain of message1 is called before a Message in the domain of message2. Each value of message1 and message2 failing to comply to this relationship is removed from the corresponding domains.

The constraint creator(classifier1, message2) (respectively created) is very similar to constraint caller(classifier1, message2), except that for each possible value of message2, there must be a corresponding value of classifier1 so that any Classifier in the domain of classifier1 is an instance of Create, and is the sourceClassifier of a Message in the domain of message2.

For example, the Memento pattern, as shown in Figure 3, is the source system and is modelled by associating a variable with each en-

tity in the scenario diagram (`var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState`, `var_aCaretaker`, `var_anOriginator`, and `var_aMemento`), and by constraining the values of these variables according to the relationships among the entities:

```

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)

```

The domain of each variable corresponds to the entities in the scenario diagram of a target system. For example, the excerpt shown in Figure 5 is a toy system in which we want to identify the Memento pattern. It involves 14 entities: `main (String[] args)`, `public void <init>()`, `public void <init>()`, `public void callCreateMemento()`, `createMemento()`, `public void <init>()`, `setState()`, `undoOperation()`, `setMemento(aMemento)`, `getState()`, `ModelMementoTest`, `Caretaker`, `Originator`, and `Memento`. The domain of variables `var_aCaretaker`, `var_anOriginator`, `var_aMemento`, and variables `var_createMemento`, `var_newMemento`, `var_setState`, `var_setMemento`, `var_getState` are respectively of size 4 and 10.

The resolution of the CSP modelling the Memento pattern returns results of the form:

```

1 <Sol.#>.var_createMemento = <an entity>
2 <Sol.#>.var_newMemento   = <an entity>
3 <Sol.#>.var_setState     = <an entity>
4 <Sol.#>.var_setMemento   = <an entity>
5 <Sol.#>.var_getState     = <an entity>
6 <Sol.#>.var_caretaker    = <an entity>
7 <Sol.#>.var_originator   = <an entity>
8 <Sol.#>.var_memento      = <an entity>

```

When applied to the toy system, our approach found one solution:

```

1 1.var_createMemento = createMemento()
2 1.var_newMemento   = new Memento()
3 1.var_setState     = setState(String state)
4 1.var_setMemento   = setMemento()
5 1.var_getState     = getState()
6 1.var_caretaker    = Caretaker [14613018]
7 1.var_originator   = Originator [12386568]
8 1.var_memento      = Memento [17237886]

```

## 6 Case Study

To evaluate our approach, we applied it on JHOT-DRAW v6.0b1 (15 KLOCs), which is a drawing editor

with a GUI based on an open source system written in Java. Although it is intentionally designed to have very clear implementations of well-known design patterns, its documentation can eventually help us determine the precision and recall properties of our approach. The scenario used to identify occurrences of the Visitor pattern in JHOTDRAW is to *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 Select the rectangle figure drawn at step 2;
4 Select the 'Cut' command from the menu;
5 Select the 'Paste' command from the menu.

```

On the one hand, when action ‘Cut’ is triggered by the user to cut a rectangle out of the document, the participant `FigureTransferCommand` calls the message `visit` on participant `AbstractFigure`. Then, `AbstractFigure` delegates the visit operation by calling message `visitFigure` on participant `DeleteFromDrawingVisitor`. Upon the completion of the visit operation, the rectangle is removed from the document by message `removeFromContainer`. On the other hand, when action ‘Paste’ is triggered by the user to paste the rectangle that was previously cut out of the document, the participant `FigureTransferCommand` calls the message `visit` on participant `AbstractFigure`. Then, `AbstractFigure` delegates the visit operation by calling message `visitFigure` on participant `InsertIntoDrawingVisitor`. Once the visit operation completes, the rectangle is inserted in the document by message `addToContainer`.

For this scenario, our approach includes *twice* in the solution variables `var_accept`, `var_visitConcreteElement`, `var_operation`, `var_objectStructure`, `var_concreteElement`, and `var_concreteVisitor`, to illustrate the actions ‘Cut’ and ‘Paste’ executed in the same scenario. By applying our approach to this scenario on a subset of JHOTDRAW (for performance issues as discussed in Section 7), we obtained four occurrences of the Visitor pattern.

Occurrence 1 is:

```

1 1.var_accept1 = visit(FigureVisitor visitor)
2 1.var_visitConcreteElement1 = visitFigure(Figure hostFigure)
3 1.var_operation1 = removeFromContainer(FigureChangeListener c)
4 1.var_objectStructure1 = FigureTransferCommand [7760420]
5 1.var_concreteElement1 = AbstractFigure [5489653]
6 1.var_concreteVisitor1 = DeleteFromDrawingVisitor [12741398]
7 1.var_accept2 = visit (FigureVisitor visitor)
8 1.var_visitConcreteElement2 = visitFigure (Figure hostFigure)
9 1.var_operation2 = setZValue (int z)
10 1.var_objectStructure2 = FigureTransferCommand [26980954]
11 1.var_concreteElement2 = AbstractFigure [31746664]
12 1.var_concreteVisitor2 = InsertIntoDrawingVisitor [2554341]

```

and Occurrence 2 is:

```

1  1.var_accept1 = visit (FigureVisitor visitor)
2  1.var_visitConcreteElement1 = visitFigure (Figure hostFigure)
3  1.var_operation1 = addToContainer (FigureChangeListener c)
4  1.var_objectStructure1 = FigureTransferCommand [26980954]
5  1.var_concreteElement1 = AbstractFigure [31746664]
6  1.var_concreteVisitor1 = InsertIntoDrawingVisitor [2554341]
7  1.var_accept2 = visit (FigureVisitor visitor)
8  1.var_visitConcreteElement2 = visitFigure (Figure hostFigure)
9  1.var_operation1 = removeFromContainer (FigureChangeListener c)
10 1.var_objectStructure2 = FigureTransferCommand [7760420]
11 1.var_concreteElement2 = AbstractFigure [5489653]
12 1.var_concreteVisitor2 = DeleteFromDrawingVisitor [12741398]

```

Since the two other occurrences are respectively mirror occurrence of Occurrence 1 and Occurrence 2, their details are not provided here. We say that Occurrence X and Occurrence Y are mirror occurrences because the variables `var_accept1`, `var_visitConcreteElement1`, `var_operation1`, `var_objectStructure1`, `var_concreteElement1`, and `var_concreteVisitor1` of Occurrence X have the same values as variables `var_accept2`, `var_visitConcreteElement2`, `var_operation2`, `var_objectStructure2`, `var_concreteElement2`, and `var_concreteVisitor2` of Occurrence Y, and vice versa.

According to the documentation in JHOTDRAW, the values of the variables provided in the Occurrence 2 and its mirror occurrence correspond to the participants and messages involved in the `Visitor` pattern. In contrast, the value of variables `var_operation1` and `var_operation2` corresponding respectively to Occurrence 1 and its mirror occurrence, `public void setZValue(int)`, are not involved in the sequence of messages corresponding to the actions ‘*Cut*’ and ‘*Paste*’. Therefore, Solution 1 and its mirror occurrence are not occurrences of the `Visitor` pattern.

## 7 Challenges and Limitations

In this section, we elaborate on several challenges we faced while reverse engineering scenario diagrams and identifying design patterns, as well as the main problems of the proposed approach.

**Dynamic and Static Analysis.** In obtaining scenario diagrams, we can choose to capture the behavior of a software system either by static analysis or dynamic analysis. Both strategies have their own drawbacks. On the one hand, even if static analysis can depict a complete picture of what could happen at runtime, it does not show what *actually* happens. Furthermore, using static analysis to retrieve dynamic data requires to analyze source code and determine the dynamic types of object references, which is not conceivable for large, complex systems [6]. On the other hand, reverse engineered scenario diagrams using dynamic

analysis represent only part of the system’s whole behavior. However, it reports precisely on the interactions between objects.

In the context of design patterns identification, precise data outweighs completeness. Therefore, we favor dynamic analysis over static analysis. To make up the incompleteness of reverse engineered scenario diagrams, we will consider as future work the merging of several traces, each reporting on one observed behavior according to one scenario (or use case). Also, using test coverage tools can help defining the scenarios that need to be executed to possibly recover all the design patterns applied during the design and implementation of a system.

### Target Language and Runtime Environment Specific Approach.

In the proposed 3-step approach for the identification of design patterns through dynamic analysis, the process of scenario diagram reverse engineering is specific to the target language: we used bytecode instrumentation to trace a software system’s method execution. This instrumentation technique has obvious drawbacks, among which it is specific to the target language, and highly coupled with a particular runtime environment. However, the fundamental principles according to which dynamic data is retrieved should not be affected. Regardless of the language (as long as it is object-oriented) or the runtime environment of the target system, a method execution is traced in such a way that instrumentation bytecode instructions are placed before and after the execution starts and ends.

In contrast, the process of design patterns identification using CSP (Figure 1 Step 3) is not specific to the target language, since its principal actors—variables, constraints, and domain—are described in terms of the constructs of the scenario diagram metamodel only.

**Scalability and Performance.** One of the key challenges while using dynamic analysis to monitor the behavior of a software system is the large amount of data traced. As the size of the target system grows, the execution trace grows in parallel, and as a result, execution time required to solve the CSP deteriorates.

Among the most commonly used abstraction mechanisms to cope with high volume of data, we used start and end markers to specify respectively the start and end of the action primary to a particular scenario. For instance, in the ‘*Cut and Paste a figure in a document*’ scenario described in Section 6, 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 chooses ‘*Cut*’ in the menu of JHOTDRAW. In the same manner, two markers are specified respectively for the start and end of action ‘*Paste*’. In this manner, method executions that are positioned outside each pair of start and end markers can be omitted from the execution trace. Results after applying our identification approach both on the original and the summarized execution trace show identical solutions for the `Visitor` pattern.

The marker mechanism is our first attempt to reduce the volume of dynamic data, and still needs some more refinements to assure that no occurrences of design pattern are omitted because some method executions are eliminated from the original execution trace.

**Design Pattern Description.** As explained in Section 3, we describe design patterns in terms of collaborations given in [5]. However, as design patterns need not be collaborating precisely as described in the Gang of Four, the design patterns description step could be automated in such a way that users could easily describe the collaborations between participants to characterize their own patterns of interest.

## 8 Related Work

The identification of design patterns in object-oriented software systems has been the subject of many works. In particular, the identification of structural design patterns has been investigated since as early as 1998 [17]. However, we are not aware of work dedicated to the identification of general non-structural design patterns. Thus, we present work related to the identification of structural design patterns, the use of dynamic data during structural design patterns identification, and the recovery of interaction diagrams.

**Structural Pattern Identification.** Wuyts [17] published a precursor work on structural design patterns identification. His approach consisted in representing systems as Prolog facts and in describing design pattern as predicates on these facts. Facts were extracted using static analysis. This approach had performance issues, could not deal with variations, and had limited precision and recall. It was followed by many other works to improve on its limits. These works include the use of constraint programming [13], explanation-based constraint programming [9], and, more recently, similarity scoring [16].

**Dynamic Data for Identification.** To the best of our knowledge, no previous work focused on the identification of behavioral and creational design patterns.

Heuzeroth et al. [10] proposed an approach that uses both static and dynamic data to identify so-called interaction patterns and exemplified their approach on the `Observer` pattern using a dedicated detection algorithm. It is unclear how this approach can be generalized to pure-behavioural/creational design patterns. Shawky et al. [15] proposed a similar approach to improve the precision and recall of a static identification approach.

Some previous works also used dynamic data in addition to structural data to improve precision and recall. In particular, most previous work on the identification of structural design patterns use data related to method calls, which can be considered as dynamic data, for example [1] or [8] used in [9].

**Recovery of Interaction Diagrams.** The recovery of interaction diagrams has been recently tackled by several authors. Briand et al. [4] proposed a method to reverse engineer UML sequence diagrams from execution traces. They used the recovered traces and a metamodel to describe UML v1.x sequence diagrams. Rountev et al. [14] described a first algorithm to reverse engineer UML v2.0 sequence diagrams by control-flow analysis. Their approach did not consider data obtained by dynamic analysis and thus is limited by the accuracy of the control-flow analysis. Briand et al. [3] introduced a complete approach to recover scenario diagrams using execution trace. Their work has inspired our own work.

## 9 Conclusion

We proposed a 3-step approach to identify behavioral and creational design patterns in source code using dynamic analysis. We described behavioral and creational design patterns in terms of scenario diagrams. Then, we reverse engineered scenario diagrams of a given software systems by means of dynamic analysis through bytecode instrumentation. Finally, we performed design patterns identification using constraint programming by identifying in the scenario diagrams of systems objects and messages conform (`caller/callee`, `follows`, and `create/created`) to the scenario diagrams of some design patterns. We evaluated our approach on JHotDraw with the `Visitor` design patterns to show its precision and recall.

Future work includes merging scenario diagrams to obtain sequence diagrams; using abstraction mechanisms that can reduce the size of execution trace without loss of data relevant to the identification of design patterns; adding new constraints and improving the CSP of the design patterns to obtain higher precision

without impacting recall; evaluating our approach on larger systems; combining this approach with a previous structural approach.

## References

- [1] 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.
- [2] Apache Jakarta Project. *Byte Code Engineering Library*, June 2006.
- [3] 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), September 2006.
- [4] Lionel Briand, Yvan Labiche, and Y. Miao. Towards the reverse engineering of UML sequence diagrams. *Proceedings of the 10<sup>th</sup> Working Conference on Reverse Engineering*, pages 57–66, November 2003.
- [5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1<sup>st</sup> edition, 1994.
- [6] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.
- [7] 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 39<sup>th</sup> conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, July 2001.
- [8] 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 19<sup>th</sup> conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–314. ACM Press, October 2004.
- [9] Yann-Gaël Guéhéneuc and Narendra Jussien. Using explanations for design-patterns identification. In Christian Bessière, editor, *Proceedings of the 1<sup>st</sup> IJ-CAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.
- [10] 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 the 6<sup>th</sup> world conference on Integrated Design and Process Technology*. Society for Design and Process Science, June 2002.
- [11] 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.
- [12] Object Management Group. *UML 2.0 Superstructure Specification*, October 2004.
- [13] 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.
- [14] Atanas Rountev, Olga Volgin, and Miriam Reddoch. Static control-flow analysis for reverse engineering of UML sequence diagrams. *Proceedings of the 6<sup>th</sup> Workshop on Program Analysis for Software Tools and Engineering*, pages 96–102, September 2005.
- [15] 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 2<sup>nd</sup> International Conference on Software Engineering*, pages 138–143, February 2005.
- [16] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros Halkidis. Design pattern detection using similarity scoring. *Transactions on Software Engineering*, 32(11), November 2006.
- [17] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In Joseph Gil, editor, *proceedings of the 26<sup>th</sup> conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.