This paper has been accepted at ASE 2001.

Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together

Hervé Albin-Amiot[*], **Pierre Cointe**,
Yann-Gaël Guéhéneuc[†], **Narendra Jussien**

École des Mines de Nantes
4, rue Alfred Kastler – BP 20823
44307 Nantes Cedex 3
France
E-mail: {albin|cointe|guehene|jussien}@emn.fr

Abstract

Design patterns ease designing, understanding, and re-engineering software. Achieving a well-designed piece of software requires a deep understanding and a good practice of design patterns. Understanding existing software relies on the ability to identify architectural forms resulting of the implementation of design patterns. Maintaining software involves spotting places that can be improved by using better design decisions, like those advocated by design patterns. Nevertheless, there is a lack of tools automatizing the use of design patterns to achieve well-designed pieces of software, to identify recurrent architectural forms, and to maintain software. In this paper, we present a set of tools and techniques to help OO software practitioners design, understand, and re-engineer a piece of software, using design-patterns. A first prototype tool, PATTERNS-BOX*, provides assistance in designing the architecture of a new piece of software, while a second prototype tool,* PTIDEJ*, identifies design patterns used in an existing one. These tools, in combination, support maintenance by highlighting defects in an existing design, and by suggesting and applying corrections based on widely-accepted design patterns solutions.*

1. Introduction

Since design patterns emerge [7], they have been widely accepted by OO software practitioners. Their contribution cover the definition, the design, and the documentation of class libraries and frameworks. They are an efficient tool to communicate, to capitalize on, and to understand solutions to common OO design problems. We do not pretend that design patterns solve on their own all the problems occurring during software designing or re-engineering, but we strongly believe that they have an important role to play.

The more important problem using design patterns is the lack of tools to automate their instantiation and detection. In this paper we propose a set of tools and techniques to help software designers use design patterns. This set of tools and techniques encompasses: (1) The choice of the right design pattern depending on the context; (2) The adaptation of the design pattern to specific requirements; (3) The application of the design pattern in a given target language; (4) The detection of complete and distorted versions of a design pattern; and, (5) The transformation of the distorted versions.

With these tools, we intend to bring a solution to the three following questions:

- While developing, how to enforce the use of design patterns?
- When reviewing, how to identify the use of design patterns?
- When maintaining, how to enforce[1] the use of design patterns?

We present a solution based on a meta-model used to formalize patterns in such a way they can be manip-

[*]This work is partly funded by Soft-Maint – 4, rue du Château de l'Éraudière – 44 324 Nantes – France

[†]This work is partly funded by Object Technology International, Inc. – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada

[1]The first and third points are alike: Enforcing the use of design patterns while developing or maintaining boils down to a unique problem: How to transform some source code such that it complies with design patterns?

ulated by tools. This meta-model serves as a basis for our two prototype tools, PatternsBox and Ptidej, which provide mechanisms to apply and to detect design patterns.

Because we believe in the usefulness of design patterns, this paper follows the formalism suggested by [7]. Each section is a pattern made of the following subsections: *Name*, *Also Known As* (*A.k.a.*), *Intent* (what is the problem we try to solve), *Motivation* (why we try to solve this problem), *Applicability* (how relevant is the solution we propose), *Consequences* (what are the consequences and the trade-offs involved by our solution), *Implementation* (how the solution is implemented), *Sample*, and *Related patterns*.

This paper is organized as follows: Section 2 introduces the meta-model, core of our tools. Section 3 presents the tool to select, adapt, and instantiate design patterns. Section 4 describes the tool to detect and correct design patterns. These tools use a Constraint Satisfaction Problem (csp) formalism [11], as presented in Section 5; an explanation-based constraint solver, as presented in Section 6; and a source-to-source transformation engine, as presented in Section 7. Finally, we summarize and conclude on our approach.

2. PDL

A.k.a. Pattern Description Language

Intent Describe design patterns as first-class entities that can be manipulated and reasoned about.

Motivations We want to reify design patterns as first-class entities providing their associated code and detecting their occurrences in existing code. We need to formalize design patterns according to these two dual requirements. It is not possible to describe formally what a design pattern is in *essence*, but the meta-modelling technique is useful to formalize design patterns in specific use-cases, like representation, application, or validation. Pdl is a meta-model that describes the semantics of a design pattern description language and which handles uniformly the instantiation and the detection of design patterns. Several meta-models for representing design patterns have already been proposed but none are specifically designed toward code generation and detection. [14] and [16] introduce meta-models for design patterns instantiation and validation but without support for code generation. In the PatternGen prototype tool [17], the meta-model does not allow design patterns detection. In [6], the fragment-based meta-model allows design patterns representation and composition.

Applicability Use Pdl to formalize and describe design patterns, and give them existence when designing a piece of software.

Consequences Pdl defines a set of meta-entities. A design pattern description is obtained by instantiation and composition of these entities. The composition follows semantic rules defined by the relationships among the meta-entities and, thus, provides a means to formalize design patterns. The result of the composition is a model, which describes the structural and behavioral aspects of the modelled design pattern. This model is not yet bound to any specific context and contains only the generic micro-architecture and intent of the design pattern. For this reason, we call this model an *abstract model* and we call the same model, refined for a specific context, a *concrete model*.

Implementation A design pattern abstract model is reified as an instance of a subclass of class `Pattern`, see Figure 1; this class inherits the services defined by the class `Pattern` and may define other services, specific to the abstract model it represents (like `addLeaf()`, for the Composite pattern abstract model). A design pattern abstract model consists of a collection of entities (instances of `PEntity`), representing the notion of participants as defined in [7]. Each entity contains a collection of elements (instances of `PElement`), representing the different relationships among entities. If needed, new entities or elements can be added by specialization of the `PEntity` or `PElement` classes. It is possible to reason about and act on design pattern abstract models as on any other regular objects, because they are represented by standard classes and instances.

Sample We use the Composite pattern from [7] (see its structure Figure 2 (Left)) and we follow the description presented under the Section *Implementation*, in paragraph *Declaring the child management operations*. This is the most common use of the Composite pattern (for example, the classes `Component` and `Container` of the Java Awt[2] follow this implementation).

The diagram in Figure 2 (Center) represents the design pattern abstract model derived from the original design pattern *leitmotiv* [5] (all the informal pieces of information given in [7] are made explicit).

Related patterns PatternsBox, in Section 3, uses Pdl to manipulate design patterns abstract models and to adapt them to a specific context. Ptidej, in Section 4, uses design pattern abstract models to detect design patterns in source code.

[2]Awt stands for Abstract Window Toolkit.

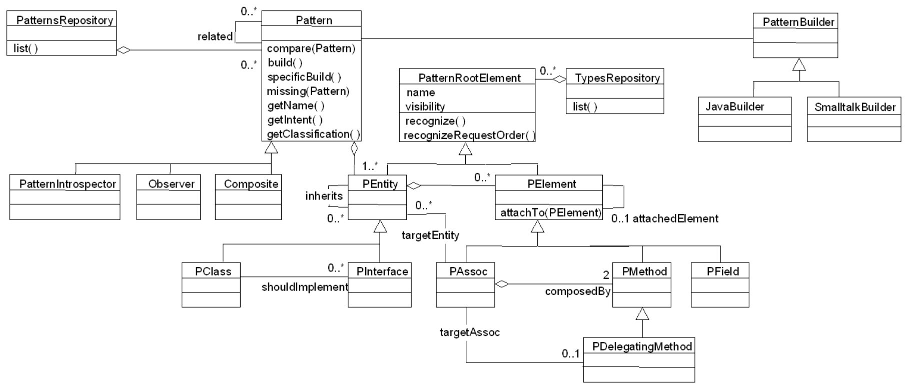


Figure 1. *A UML-like partial representation of our meta-model.*

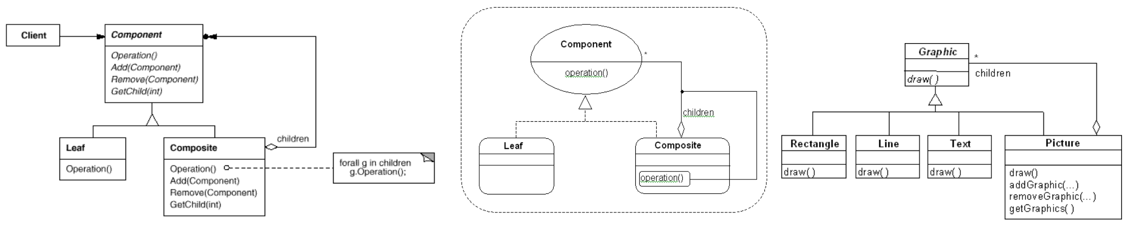


Figure 2. **(Left)** *The OMT-like diagram of the Composite pattern from [7].* **(Center)** *A representation of the Composite pattern abstract model. The dotted rounded square is an instance of the class `Composite`, subclass of class `Pattern`. Rounded squares represent instances of `PClass`; ovals, instances of `PInterface`; and, boxed names, instances of `PDelegatingMethod`. Instances of `PAssoc` and realization links use the standard UML notation.* **(Right)** *UML diagram of the concrete model.*

3. PatternsBox

A.k.a. DESIGN PATTERNS OUT-OF-A-BOX

Intent Provide an intuitive way to select, adapt, and apply a design pattern.

Motivations The application of a design pattern can be decomposed in three distinct activities: (1) The choice of the right design pattern, which fulfils the user requirements; (2) Its adaptation to these requirements (the term *instantiation* is commonly used to identify this task); and (3) The production of the code required for its implementation. PATTERNSBOX is a tool designed to help developers accomplish these three tasks.

Applicability Use PATTERNSBOX to select the required design pattern; to adapt the associated abstract model defined using PDL; and, finally, to generate the corresponding source code.

Consequences PATTERNSBOX provides:

- Access to the design pattern repository and, for each design pattern, a shortcut to its intent.
- Mechanisms to adapt a selected design pattern abstract model to a given context, thus creating a concrete model of the design pattern.
- Generation of the source code associated with the design pattern concrete model. Our tools shall be able to modify existing source code to instantiate design patterns, using a source-to-source transformation engine.

Implementation Currently, navigation through the design patterns repository is guided using the Html version of [7]. We plan to integrate a more sophisticated mechanism such as in [13]. Once an abstract model is selected, PATTERNSBOX uses Java introspection capabilities to discover its properties, manipulate, and adapt it. Then, PATTERNSBOX dialogs through message-sends with instances of the abstract model. These messages may be those understood by the `Pattern` class (like `build()`, to generate code) or those specific to the abstract model.

Sample The following example has been introduced in [7] and reused in [14]. It defines a hierarchy of graphical components as shown Figure 2 (Right).

The following lines of code declare a new Composite concrete model. PatternsBox performs these operations automatically.

```
Composite p = new Composite();
p.getActor("Component").setName("Graphic");
p.getActor("Component")
    .getActor("operation").setName("draw");
p.getActor("Leaf").setName("Text");
p.getActor("Composite").setName("Picture");
p.addLeaf("Line");
p.addLeaf("Rectangle");
```

From this new concrete model, the source code generated by PatternsBox is:

```
/* Graphic.java */
public interface Graphic {
    public abstract void draw();
}
/* Picture.java */
public class Picture implements Graphic {
    // Association: children
    private Vector cildren = new Vector();
    public void addGraphic(Graphic aGraphic) {
        children.addElement(aGraphic);
    }
    public void removeGraphic(Graphic aGraphic) {
        children.removeElement(aGraphic);
    }
    // Method linked to: children
    public void draw() {
        for(Enumeration enum = children.elements();
            enum.hasMoreElements();
           ((Graphic) enum.nextElement()).draw());
    }
}
/* Text.java */
public class Text implements Graphic {
    public void draw() {}
}
/* Line.java */
public class Line implements Graphic {
    public void draw() {}
}
/* Rectangle.java */
public class Rectangle implements Graphic {
    public void draw() {}
}
```

Related patterns PatternsBox uses Pdl, from Section 2, to navigate through the design pattern repository, to refine abstract models, and, finally, to generate the source code.

4. PTIDEJ

A.k.a. Pattern Trace Identification, Detection, and Enhancement for Java

Intent Automate the identification of design patterns complete and distorted versions in source code and refactor the distorted ones to comply closely with the corresponding design pattern.

Motivations It is difficult to use design patterns upfront when designing an application. When design patterns are used in a software architecture, their implementation often blurs the relations between (on the one hand) the design patterns actors and their relationships and (on the other hand) the software classes and their relationships. The semantic gap between design and implementation increases the difficulty of the overall maintenance task and of the identification of design patterns.

Many studies address the problem of automating the identification of design patterns in source code, like [1, 2, 4, 10, 15]. However, these approaches only consider complete implementations and ignore the refactoring aspects.

Applicability Use Ptidej when you want to identify design patterns in the source code of an application. A pattern is represented by an abstract model. At the source code level, an abstract model is translated into a micro-architecture. Then, we say that a micro-architecture is a *complete* version of a design pattern, if it strictly complies with the pattern abstract model micro-architecture; and, we say that a micro-architecture is a *distorted* version, if some relationships are missing. If Ptidej finds distorted versions of a design pattern, it computes the differences between them and the design pattern abstract model micro-architecture and suggests corrections to decrease these differences. These corrections are performed by a source-to-source transformation engine.

The same techniques and tools can be used to detect and correct inter-class design defects [9].

Consequences Ptidej makes it possible to:

- Reference a design pattern. We model a design pattern as an abstract model and the tool generates the corresponding csp.
- Load and display an application architecture.
- Produce a simplified model of the application architecture.
- Solve the csp with the simplified model as domain, using an explanation-based constraints solver. By solving this csp, Ptidej identifies a design pattern micro-architecture complete and distorted versions (and the associated transformation rules) in a software architecture.
- Visualize the complete and distorted versions of a design pattern (the csp solutions).
- Select a specific distorted version.
- Transform the parts of the application source code corresponding to the selected distorted version, such

that the source code better complies with the corresponding design pattern abstract model micro-architecture.

- Load the modified source code.
- Display the new application architecture using a UML-like notation.

Implementation We use the Java programming language and different existing libraries to implement PTIDEJ. We use PDL to model source code. Because there is a duality between Java source code and Java byte-codes, we use CFPARSE [8] to analyse Java classes and build a model of these classes based on our meta-model. We display this model using a graphical framework dedicated to our meta-model. From the model of the Java classes, PTIDEJ generates a new model in the Claire programming language [3], a language used to express the CSP.

Sample We apply the constraints defined for the Composite pattern on the source code of the application Figure 3 (Left). The constraint solver generates the set of all the groups of entities similar to the Composite pattern abstract model. These groups are visible in Figure 3 (Center) as the grey boxes outlining the classes. The group of entities `Element`, `Paragraph` and `Document` is one example. The greater the number of constraints relaxed, the less similar to the Composite pattern is the solution group and the lighter are the outlining boxes.

The grey-shaded boxes represent the entities belonging to (at least) one group of entities similar to the Composite pattern. When a box is selected, it highlights all the entities belonging to this particular group and presents related information: The degree of similarity of the group with the original abstract model, the constituents of this group, their values, and the associated transformation rules. The group composed of classes `Element`, `Document` and `Paragraph` is similar to the Composite pattern at 50 percent. The transformation to apply is given by the `XCommand` field:

```
Composite, Component
  | javaXL.XClass c1, javaXL.XClass c2
  | c1.setSuperclass(c2.getName());
```

This means that the class playing the role of `Composite` must be a subclass of the class playing the role of `Component`. In the example, the class `Document` must be a subclass of `Element`. The transformation engine automatically performs the modifications on the application source code by executing the `XCommand`. Figure 3 (Right) illustrates the resulting architecture of the application.

Related patterns PTIDEJ uses abstract models, defined in Section 2. It translates a design pattern abstract model into a CSP, presented in Section 5. The explanation-based constraint solver in Section 6 solves the CSP. The transformation engine, introduced in Section 7, transforms the application source code.

5. CSP

A.k.a. CONSTRAINT SATISFACTION PROBLEM

Intent Define the problem of detecting a design pattern, in terms of its variables, the constraints among them, and their domains.

Motivations We deduce a CSP from a design pattern abstract model and a given source code. This CSP represents the problem our explanation-based constraint solver, PALM, solves to identify, in the given source code, micro-architectures that are identical or similar to the micro-architecture defined by a design pattern.

Applicability Use a CSP when you want to resolve a problem defined by the values to obtain rather than by the process to obtain these values.

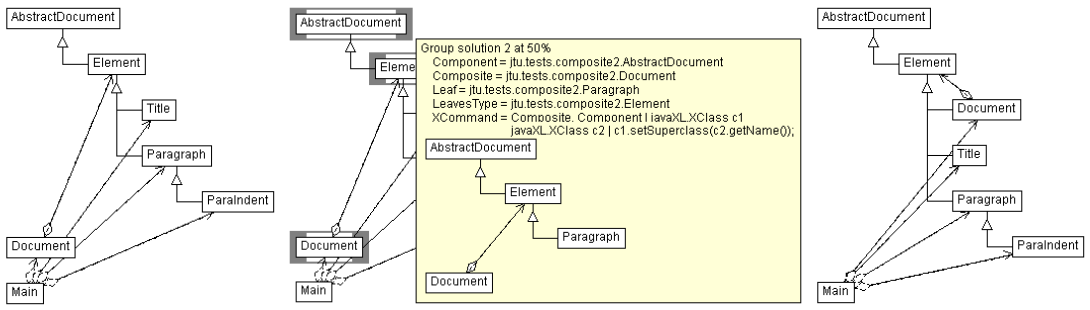
The CSP is divided into three parts:

- The set of the variables. This set corresponds to the entity[3] constituting a given design pattern abstract model.
- The set of the constraints among the variables. This set corresponds to the relationships among the entities defined by the given design pattern abstract model.
- The domain of the variables. This set corresponds to the entities of the given source code and their relationships.

Consequences Using PALM, the CSP solution is:

- The set of all the complete solutions to the problem. A complete solution is a set of entities from the given source code whose relationships satisfy the constraints of the problem.
- A subset of all the distorted solutions to the problem. A distorted solution is a set of entities from the given source code whose relationships satisfy a subset of the problem constraints. The choice of the subset is performed either interactively by the user or programmatically by associating weights with each constraint.

[3]If we assume a Java-like object-oriented programming language, an entity may be either a class or an interface.



```
Group solution 2 at 50%
   Component = jtu.tests.composite2.AbstractDocument
   Composite = jtu.tests.composite2.Document
   Leaf = jtu.tests.composite2.Paragraph
   LeavesType = jtu.tests.composite2.Element
   XCommand = Composite. Component | javaXL.XClass c1
                    javaXL.XClass c2 | c1.setSuperclass(c2.getName());
```

Figure 3. **(Left)** *The architecture of the application.* **(Center)** *A suggestion of modification.* **(Right)** *The modified architecture.*

Implementation A CSP is expressed using the CLAIRE programming language [3], in which the PaLM explanation-based constraint solver is written. The CSP is automatically obtained from the design pattern abstract model.

Sample The Composite pattern, as presented in Figure 2, is modelled by associating a variable with each defined entity (`Component`, `Composite` and `Leaf`), and by constraining the values of these variables according to the relationships among the entities: `composite` $<$ `component`, `leaf` $<$ `component`, and `composite` $\supset$ `component`.

The source code of the application, Figure 3 (Left), involves seven entities: `AbstractDocument`, `Element`, `Title`, `Paragraph`, `ParaIndent`, `Document`, and `Main`. The domain of each variable of the CSP, `component`, `composite`, and `leaf`, is of size 7 (one slot for each possible entity from the source code).

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

```
<Sol.#>.<Quality>.component = <an entity>
<Sol.#>.<Quality>.composite = <an entity>
<Sol.#>.<Quality>.leaf = <an entity>
```

A solution of weight 50, without constraint `Component` $<$ `Composite`, see Figure 3 (Center), is :

```
1.50.component = Element
1.50.composite = Document
1.50.leaf = Paragraph
```

Related patterns We use a given design pattern abstract model as presented in Section 2, to define a CSP. The explanation-based constraint solver presented in Section 6 solves this CSP. The PTIDEJ tool defined in Section 4 generates the domain of the CSP, calls the constraint solver, and displays the results.

6. PaLM

A.k.a. PROPAGATION AND LEARNING WITH MOVE

Intent Define a new kind of constraint solver capable of providing explanations and information about its actions (including explanations for a lack of solutions to a given problem).

Motivations Complete solutions to the CSP modelling our problem are of no use when trying to understand and to improve source code. What is really interesting are distorted solutions: Solutions that do not respect all the constraints defining a complete solution. Constraints that are not verified point out what should be improved in the analyzed code.

A commonly used approach [19] is to *a priori* identify the possible distorsions and add them to the set of examples defining the design pattern. This can become extremely hard to maintain when adding new design patterns or modifying old ones.

Classical constraint solvers are not designed to compute distorted solutions. They are only capable of computing complete solutions or of telling that there is no solution at all. No commercial system can automatically provide insights on distorted solutions to a given problem. Thus, we need to define a new solver capable of handling the search of distorted solutions.

Applicability We use a new paradigm in constraint programming: Explanation-based constraint solving [11]. The idea is to maintain information (namely explanations), through the search, about (direct or indirect) effects of constraints appearing in the system. Explanations are an abstraction of the trace of a constraint solver.

Explanations can be used:

- To point out which set of constraints is responsible for the current solution (if one exists).

- To point out which set of constraints is responsible for a contradiction (if no solution exists).
- To remove incrementally a constraint by undoing dependent actions instead of a complete re-execution, as it is done by classical solvers.

The last two features make explanation-based constraint programming an effective candidate to bring an answer to our intent.

Consequences When using explanation-based constraint programming, the reason why no (more) instances of the searched design pattern exist can be clearly stated to the developer: The solver gives the set of constraints justifying that situation.

From that explanation, constraints can be removed incrementally leading to distorted solutions. The set of removed constraints is a description of why this distorted solution is a quasi-solution and therefore this set points out where to improve the source code.

Our tool can be automated if a set of preferences is associated with the set of constraints defining the searched design pattern. Solutions provided by the automated and user-driven versions are the same. The interactive version allows the user to control completely the search and to focus precisely on the distorted solutions that are of interest.

Implementation Explanations are computed by keeping a limited (polynomial space occupation) trace (polynomial time computation) of the behavior of the solver. We use the PaLM system [11], an explanation-based constraint system built on top of an open-source constraint solver, Choco [12] (both implemented using the Claire programming language [3]).

Sample Let us consider a two-variable toy problem: x and y with the same set of possible values $[1, 2, 3]$. Let us state $x > y$. The resulting sets of possible values are $[2, 3]$ for x and $[1, 2]$ for y. An explanation for this situation is the constraint $x > y$.

Let us suppose that we choose to add the constraint $x = 2$. The resulting possible value for x is only 2. The explanation for the modification is the constraint $x = 2$. The other consequence is that the remaining value for y is 1. The explanation for this situation is twofold: A direct consequence of the constraint $x > y$ and also an indirect consequence of constraint $x = 2$.

Related patterns Ptidej, the tool dedicated to the detection of design patterns presented in Section 4, uses PaLM to solve the CSP defined in Section 5.

7. JavaXL

A.k.a. Java eXtended Language

Intent Modify user source code to comply with a design pattern description.

Motivations Applying a design pattern on an existing implementation requires to modify the corresponding source code as little as possible. If the developer uses specific idioms, coding conventions, or comments, it is mandatory to preserve them. For example, if a design pattern description implies that the visibility of a field must be private, then only the place where the modifier is declared in the source code must be changed. Existing source-to-source transformation engines dedicated to the Java programming language, like OpenJava [18], do not take this aspect into consideration and perform a complete source re-generation.

Applicability JavaXL can be used to apply design patterns and to apply specific code conventions or idioms. JavaXL ensures that the resulting source code is syntactically correct but does not ensure that it is semantically correct (compilable). The transformation-rule writer is in charge of the transformation semantics.

Consequences JavaXL ensures that:
Let S be an original source code
Let S' be a translated source code
Let $T(S) \rightarrow S'$ be a transformation function
Then: $T^{-1}(T(S)) = S$

Implementation JavaXL is an extension to the Java reflection API, which works at runtime and provides introspection mechanisms. JavaXL is intended to work during source code edition. This is a major difference with existing source-to-source transformation engines (such as OpenJava), which perform, before compilation, translations that are not intended to be readable, only compilable. JavaXL provides a class hierarchy equivalent to the Java reflection API. This hierarchy provides read access (introspection) and write access (intercession) on every language entities: Classes, fields, methods...

Sample Let us take the example presented in Section 4 where an entity `Document` must be declared as a sub-entity of an other entity `Element`. Let the following source code be the source code for `Document`:

```
public class Document
    /* Here is an important comment */
    implements Cloneable {
    ...
}
```

The following code:

```
XClass document = new javaXL.XClass("Document");
document.setSuperclass("Element");
```

produces the following result:

```
public class Document extends Element
    /* Here is an important comment */
    implements Cloneable {
    ...
}
```

Related patterns PTIDEJ, in Section 4, uses JAVAXL to perform source transformations to make a given source code compliant with a given design pattern description. PATTERNSBOX shall use JAVAXL as well. So far, it does not use the source-to-source transformation engine because it is not dedicated to a specific programming language. We must first adapt JAVAXL to other languages, in addition of Java (like C++ or Smalltalk).

8. Conclusion

In this paper, we showed how to solve the problem of automating the instantiation and the detection of design patterns. We have presented a set of tools and techniques that, put together, bring a solution to this problem. We presented synthetically, as patterns, the different tools and techniques needed to solve this problem. These techniques encompass:

- A meta-model, PDL, to describe design patterns with an instantiation- and a detection-centric view. Using this meta-model, we can express design pattern as first-class entities that embody all the needed information for instantiation and detection.
- A source-to-source transformation engine, JAVAXL, to instantiate the design patterns, while modifying as little as possible the user's source code.
- A CSP to define the rules for design pattern detection.
- An explanation-based constraint solver, PALM, to solve the CSP.

We used these techniques to define two prototype tools that assist the developers in instantiating and detecting design patterns:

- PATTERNSBOX uses PDL to manipulate and adapt design patterns models, to generate source code. In the future, it shall useJAVAXL to transform existing source code.
- PTIDEJ uses PDL to manipulate design patterns models, a CSP and the PALM constraint system to detect complete and distorted versions of design patterns, and JAVAXL to make the distorted versions compliant with the design pattern models.

We are currently assessing the usability and suitability of our tools on several frameworks: Java AWT, JUNIT, JHOTDRAW, and JEDIT.

Our proposal offers one answer, based on design patterns, to automate, or to assist, in designing, understanding, and re-engineering software. In the future, we hope to see other approaches to this problem, based on the patterns we have presented.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. *Proceedings of the 6 th Workshop on Program Comprehension*, pages 153–160, 1998.

[2] K. Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

[3] Y. Caseau and F. Laburthe. Claire: Combining objects and rules for problem solving. *Proceedings of JICSLP, workshop on multi-paradigm logic programming*, 1996.

[4] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. *Proceeding of TOOLS*, 30:18–32, 1999.

[5] A. H. Eden. *Precise Specification of Design Patterns and Tool Support in their Application.* PhD thesis, Tel Aviv University, 2000.

[6] G. Florijn, M. Meijers, and P. V. Winsen. Tool support for object-oriented patterns. *Proceedings of ECOOP*, 1997.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[8] M. Greenwood. *CFParse Distribution.* IBM AlphaWorks, September 2000.

[9] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. *Proceedings of TOOLS USA*, 2001.

[10] J. Jahnke and A. Zündorf. Rewriting poor design patterns by good design patterns. *Proceedings the Workshop on Object-Oriented Reengineering at ESEC/FSE*, September 1997.

[11] N. Jussien and V. Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, Sept. 2000.

[12] F. Laburthe. CHOCO's API. Technical Report Version 0.13, OCRE Committee, 2000.

[13] O. Motelet. An intelligent tutoring system to help OO system designers using design patterns. Master's thesis, Vrije Universitët, 1999.

[14] B.-U. Pagel and M. Winter. Towards pattern-based tools. *Proceedings of EuropLop*, 1996.

[15] L. Prechelt and C. Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, 1998.

[16] P. Rapicault and M. Fornarino. Instanciation et vérification de patterns de conception : Un méta-protocole. *Proceedings of LMO, in French*, pages 43–58, 2000.

[17] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel. Design patterns application in UML. *Proceedings of ECOOP*, 2000.

[18] M. Tatsubori. An extension mechanism for the Java language. Master's thesis, Graduate School of Engineering, University of Tsukuba , Ibaraki, Japan, 1999.

[19] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.