Using explanations for design patterns identification

Yann-Gaël Guéhéneuc and **Narendra Jussien**
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3
{Yann-Gael.Gueheneuc|Narendra.Jussien}@emn.fr

Abstract

Design patterns describe micro-architectures that solve recurrent architectural problems in object-oriented programming languages. It is important to identify these micro-architectures during the maintenance of object-oriented programs. But these micro-architectures often appear distorted in the source code. We present an application of explanation-based constraint programming for identifying these distorted micro-architectures.

1 Introduction

The production of quality source code is an important matter for the software industry. A quality source code facilitates evolutions and maintenance: Addition of new functionalities, bug corrections, adaptation to new platforms, integration with new class libraries.

In object-oriented programming, a quality source code has two aspects: Efficient and clearly-written algorithms, respecting conventions and idioms, and an *elegant* class architecture. If many works studied the former aspect ([Demeyer *et al.*, 2000] gives a synthesis), the latter aspect is more difficult to define and has not been often studied (we can mention [Jahnke and Zündorf, 1997]).

A micro-architecture describes the structure of a subset of the classes[1] of an object-oriented program. The solutions proposed by the *design patterns* [Gamma *et al.*, 1994] are examples of **good** micro-architectures. However, it is not easy to write directly source code that carefully respects design patterns. Thus, most of the time, only *distorted* design patterns are present in the source code (*i.e.*, micro-architectures similar to – not identical to – those proposed by the design patterns). There is a lack of tools helping to identify distorted versions of design patterns in existing source code and indicating possible improvements by pointing out differences with the *exact* design pattern.

In this article, we propose the use of explanation-based constraint programming [Jussien and Barichard, 2000] to identify and to correct micro-architectures similar to design patterns. Section 2 introduces the notion of design pattern more precisely. Section 3 recalls the solutions proposed in the literature for the identification and correction of program source code. Section 4 discusses notions on explanations for constraint programming. Section 5 presents our solution to identify architectures similar to design patterns and Section 6 gives some results obtained on industrial libraries.

[1] For the sake of clarity, in the remainder of this article, we use the object-oriented programming language JAVA: The term *class*, in particular, indifferently represents a class, an abstract class or an interface.

2 Design patterns

The architecture of a software system is unique. It depends on the context in which it is developed and on various aspects: Expected lifetime, cost of development, foreseen evolutions, experience of architects and developers, ... For a particular problem, there does not exist **one** optimal architecture but rather an architecture adapted to a given context. Therefore, we focus on general and context-independent architectural problems.

2.1 Definition

For recurring architectural problems, *design patterns* [Gamma *et al.*, 1994] represent solutions that are independent of the context and of the object-oriented language. They capture the experience of skillful developers. A definition of a design pattern contains four essential parts:

1. A unique name identifying the pattern
2. A description of the addressed architectural problem
3. A solution to the problem, with class diagrams representing the involved classes and their roles (using an OMT-like notation – Object Modelling Technique [Rumbaugh *et al.*, 1991])
4. The consequences and possible trade-offs of the solution

For example, The Composite design pattern builds complex structures of objects by composing recursively objects of same nature in a tree-like manner. It lets treat uniformly objects and compositions of objects. The general structure of the Composite design pattern is shown Example 1. The Composite design pattern definition presented in [Gamma *et al.*, 1994], pp. 163–173, embodies eleven sections over ten pages (ten pages is the average length of a design pattern definition in this book). The first three sections, *Intent*, *Motivation* and *Applicability*, introduce the problem (part 2): How to compose objects into tree structures representing part–whole hierarchies, and how to treat uniformly individual objects and compositions of objects. The three next sections, *Structure*, *Participants* and *Collaborations*, propose a solution (part 3) to the problem with class diagrams, instance diagrams, a list of participants (*i.e.*, a list presenting the class roles), and a list of collaborations (*i.e.*, a list presenting the class interactions). The *Consequences* section (part 4) states the effects of applying this design pattern : Simplification of the clients and of the implementation. Finally, the last four sections, *Implementation*, *Sample Code*, *Known Uses* and *Related Patterns*, provide specific information about the implementation and the use of the solution.

The description of the problem solved by a design pattern is always informal. It represents more a set of motivations for which to apply this design pattern than a precise description of the addressed problem. We cannot directly use this description to detect architectural problems. On the other hand, we can consider the architectural solution, described with class diagrams and source code examples, as an example of a **good** micro-architecture.

> **Example 1 (An overview of the Composite design pattern) :**
> The general structure of the Composite design pattern is shown Figure 1. The abstract class (or interface) `Component` defines an `operation()` that is indifferently applied on an object of type `Leaf` or on a composition of objects of type `Component`, `Composite`. The instances of the `Composite` class are in charge of applying `operation()` on their children.

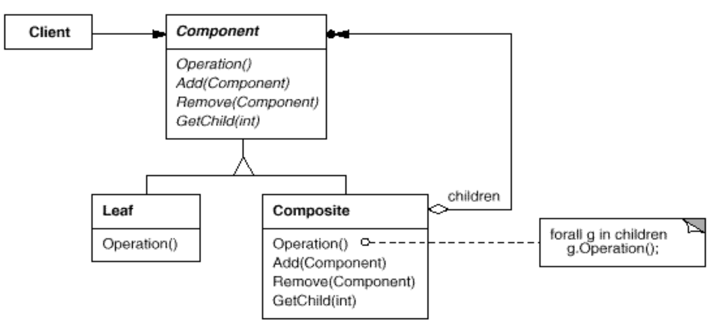

Figure 1: The Composite design pattern

2.2 Towards a higher quality code

A software architecture is subject to evolutions and transformations during its life cycle. These evolutions and transformations slow the use of design patterns or impede their quality. It is difficult to apply *a priori* design pattern solutions – good micro-architectures – when the software is not finished yet. Applying design patterns requires a thorough knowledge of all the existing design patterns – knowledge that only a few developers possess –, and insight on the overall architecture. Thus, we propose to use design patterns to improve source code quality rather than to produce **directly** design pattern compliant source code.

We want to identify places in the architecture where quality would be improved by the introduction of design patterns. Our approach consists in detecting groups of classes, whose structures are close to a micro-architecture solution suggested by a design pattern (see Example 2). The class structure of such a group could be improved by applying the solution of the design pattern. Our approach consists: **(a)** in a description of the relationships among classes introduced by a design pattern; **(b)** in using an explanation-based constraint solver for detecting, in the source code, classes whose structure is close to a (already referenced) design pattern; and **(c)** in transforming the source code accordingly to the design pattern specifications.

In this article, we describe the **(b)** phase, the identification phase.

> **Example 2 (Identification of the Composite design pattern) :**
> Let us consider the development of an application to produce representations of documents. In Figure 2 on the left, the kernel of the application consists of a class `Element` that defines a common interface for all the elements of a document: Titles (class `Title`), paragraphs (class `Paragraph`), and indented paragraphs (class `ParaIndent`), ... A `Document` class composes those elements to describe a document.
> This architecture is similar to the Composite design pattern. But the specifications of the design pattern require that the `Document` class be a subclass of the `Element` class, to unify their interfaces and to allow the composition of documents (Figure 2, on the right).

3 State of the Art

In software engineering and re-engineering, only few studies exist on automating the identification and the correction of design defects. There are two reasons for this lack of material. On one hand, the automation of processes and techniques related to an *intellectual* activity, such as software development, is not welcomed. This automation is not welcomed because of the seeming loss of control resulting from the automation of a process acting on software – software which is already difficult to maintain. This loss of control from the automatic detection and correction of design defects is perceived as too important compared to the benefits brought by the automation. On the other hand, when solutions have been proposed, these solutions were reduced to the problems of (semi-) automatically detecting or correcting design defects of the classes themselves [Jahnke and Zündorf, 1997; Fowler, 1999; Demeyer *et al.*, 2000] (for example, problem of long methods or lack of cohesion among the methods of a class); or to the problems of detecting design patterns in existing source code to help documenting or understanding legacy systems [Brown, 1996; Krämer and Prechelt, 1996; Wuyts, 1998; Mancoridis *et al.*, 1998; Richner and Ducasse, 1999]. Those works clearly show that a fully automatic approach is too ambitious because of the software complexity: The user input is really important.

A few techniques exist to identify design patterns. Among those techniques, the use of logic programming is of great interest [Wuyts, 1998]: A design pattern is described as a set of

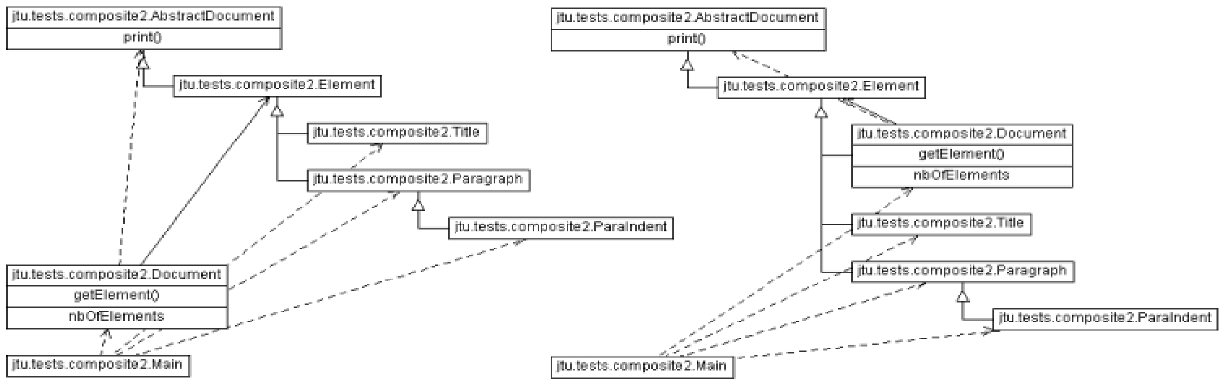

Figure 2: Kernel of the application to describe documents: On the **left** the original architecture and on the **right** the corrected architecture. A class is depicted as a box – with one ore more division – containing the class name – and possibly the class associations, methods, and fields. An association is represented as a plain arrow from the aggregate class to its component. Dotted arrows are instance creation and knowledge links. A square line with an empty triangle corresponds to inheritance.

logical rules. The logical rules unify with the facts representing the source code to identify design patterns. But this technique is limited. It only detects classes whose relationships are described by the logical rules. It does not directly allow to detect similar rather than identical micro-architectures of a design pattern. The rules must be extended to introduce the missing distorted cases to obtain more solutions. Consequently, the rules become quickly impossible to manage. The addition of new design patterns requires thinking about all the possible distortions when conceiving the system of rules.

Outside the *object-oriented languages* community, the search for sub-graphs in a graph [Régin, 1995] presents similarities with our work. But, to our knowledge, the search of sub-graphs *similar to* and not merely identical to a given sub-graph has not been studied yet. Another related work is the phase of *adaptation* in case-based reasoning. [Fuchs *et al.*, 2000] has recently presented a technique adapted to continuous domains (with an order relation on the values) but this technique is unadapted to our discrete problem.

To conclude, an acceptable solution to our problem must favor a dialog with the developer:

- To **explain** concretely why the architecture of a group of classes is a distorted version of an existing design pattern
- To direct dynamically the search of such architectures by proposing **interactively** the exclusion of such or such characteristic of the design pattern (to avoid determining *a priori* the possible evolutions)

These are the reasons why we propose the use of explanations-based constraint programming.

4 Explanation-based constraint programming

Explaining and suggesting possible architectural modifications is an interesting way to improve object-oriented source code. Explanation-based constraint programming already proved its interest in many applications [Jussien and Barichard, 2000]. We recall in this section what it is and how it can be used.

4.1 Explanations

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

A **contradiction explanation** (*a.k.a.* **nogood** [Schiex and Verfaillie, 1994]) 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 divides 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.

$$C \vdash \neg\,(C' \wedge v_1 = a_1 \wedge ... \wedge v_k = a_k) \qquad (1)$$

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]:

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

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:

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

[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. By doing so, the space complexity remains polynomial. We keep only **one** explanation at a time for a value removal.

4.2 Computing explanations

Minimal (*w.r.t.* inclusion) explanations are the most interesting. They allow very precise information on emerging dependencies among variables and constraints, dependencies identified during the search. Unfortunately, computing such explanations is time-consuming [Junker, 2001]. A good compromise between size and computability is the use of the knowledge that is *inside* the solver. Indeed, constraint solvers always know (although not often explicitly) why they remove values from the domains of variables. Precise and interesting eliminating explanations can be computed by explicitly stating such information. To achieve this behavior, it is necessary to alter the solver code itself. [Jussien and Barichard, 2000] is an introduction to modifying the solver.

4.3 Using explanations

Explanations can be used in several ways [Jussien *et al.*, 2000; Jussien and Barichard, 2000; Jussien and Lhomme, 2000]. Debugging purposes pop to the mind: To explain **clearly** failures, to explain differences between intended and observed behavior for a given problem (why is value 4 not assigned to variable x ?).

Explanations can be used also to determine direct or indirect effects of a given constraint on the domains of the variables of the problem, and for dynamic constraint removal. This is the case with the justification system used in [Bessière, 1991] 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 facilitates the building of dynamic over-constrained problem solver [Jussien and Boizumault, 1997].

Less direct applications are possible 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 to improve standard backtracking and to exploit information gathered to improve the search: To provide intelligent backtracking [Guéret *et al.*, 2000], to replace standard backtracking with a jump-based approach *à la* Dynamic Backtracking [Ginsberg, 1993; Jussien *et al.*, 2000], or even to develop new local searches on partial instantiations [Jussien and Lhomme, 2000].

But, what is interesting in the design pattern identification context is the ability of explanation systems:

- To explain why no solution is found to 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. The interested reader should refer to [Jussien and Barichard, 2000] for more information.
- To provide a data-driven search (*i.e.*, through the user input): A contradiction explanation justifies the lack of more solutions for the current problem. Selecting and relaxing a constraint given by the explanation allows the discovery of *new* solutions (distorted solutions of the original problem). The selection is left to the user who knows which constraint to relax to keep the *principle* of the design pattern being searched. Data-driven search of design patterns is detailed in Section 5.2.

[3] A nogood is said to be relevant if all the decision constraints in it are still valid in the current search state [Bayardo Jr. and Miranker, 1996].

5 Application to the problem

The detection of micro-architectures similar to a design pattern using explanation-based constraint programming consists:

1. In modelling a set of design patterns as CSP: The micro-architecture solution proposed by a design pattern is modelled as a set of constraints. A variable is associated with each class defined by the design pattern. The variables of our model are integer-valued. The domain of a variable is the set of all the existing classes in the source code. Each class is identified by a unique integer. The relationships among classes (inheritance, association, *etc.*) are represented by constraints over the variables. See Example 4.
2. In modelling the user's source code to keep only the information needed to apply the constraints: The class names – forming the domain of the variables –, and the relationships among classes – abstracted in tables attached to the library of constraints. See Example 5.
3. In resolving the CSP to search distorted solutions – solutions that violates one or several constraints specified by the design pattern: When all the real solutions of the CSP are found, the search is guided dynamically by the user to find interesting distorted solutions. Information (explanations of contradiction) provided by the constraint solver help the user.

5.1 A library of specialized constraints

From the relationships among classes defined in [Gamma *et al.*, 1994], we built a first library of constraints. Specialized constraints express the inheritance, association, knowledge, *etc.* relationships. These constraints involve variables representing one and only one class 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. The propagation mechanism ensures the consistency of the variable domains because of the specific nature of our constraints.

Our library offers constraints covering a broad range of design patterns. However, some design patterns are difficult to express and need additional relationships or the decomposition of some relationships into sub-relationships.

We provide the following symbolic constraints in our library (these constraints can be combined to form more complex constraints):

- **Strict inheritance**: Establishes an inheritance relationship between two classes – between two variables – such as defined in Example 3. This relationship is enforced by the `StrictInheritanceConstraint`. From this notion of strict inheritance, we derive the notion of inheritance (`InheritanceConstraint`) such as $\mathtt{A} < \mathtt{B}$ or $\mathtt{A} = \mathtt{B}$ (the two variables may represent the same class).

> **Example 3 (Strict-inheritance constraint) :**
> An inheritance relationship links two classes in a parent–child-like relationship – *i.e.*, superclass–subclass. When considering single inheritance, the inheritance relationship is a partial order, denoted $<$, on the set of classes E. For any pair of distinct classes A and B in E, if B inherits from A then: $\mathtt{A} < \mathtt{B}$.
> The constraint associated with the inheritance relationship is a binary constraint between two variables (classes) A and B. The operational semantics of this constraint is: (d_X represents the domains of variable X)
>
> $$\forall class_A \in d_A, \exists class_B \in d_B, class_A < class_B$$
>
> $$\forall class_B \in d_B, \exists class_A \in d_A, class_A < class_B$$

- **Knowledge**: Establishes a knowledge relationships between two classes. The class A knows about the class B if methods defined in A invoke methods of B. This relationship is binary, oriented, and not transitive. We denote this relationship by $\mathtt{A} \rhd \mathtt{B}$ and the constraint `RelatedClassesConstraint` enforces it.
- **Non-knowledge**: Ensures a *reciprocal* relationship. The class A must not know about the class B. The constraint `UnRelatedClassesConstraint` expresses this relationship.
- **Composition**: Ensures that two classes are composed. A class A is composed with instances of a class B if the class A defines one or more fields of type B. We write $\mathtt{A} \supset \mathtt{B}$. This relationship is binary, oriented, and not transitive. It is enforced by the constraint `CompositionConstraint`.
- **Field type**: Ensures that the field f of class A is of type B: $\mathtt{A.f} = \mathtt{B}$. The constraint `PropertyTypeConstraint` establishes this link. We use this generic constraint to define easily new constraints.

5.2 Behavior of the solver

Our dedicated constraint library is not sufficient to solve our problem. Indeed, solutions that fit exactly in the definition of a design pattern (the *real* solution of the CSP) are of no use to improve the quality of the user's source code. We need to find *distorted* solutions – assignments that do not verify all the features of the intended pattern– *i.e.*, that violates at last one of the constraints defining the design pattern. Explanation-based constraint programming is a key tool for solving that problem.

First, real solutions are computed. This computation ends by a contradiction (there is no more solution). Explanation-based constraint programming provides a contradiction explanation for this failure: The set of constraints justifying that any other combination of classes do not verify the constraints describing the design pattern. We do not need to relax other constraints than the constraints provided by the contradiction explanation: We would find no other real solutions. A contradiction explanation provides insights on which distorted solutions are available – more precisely, on which distortions (constraint violations) would lead to more results, if the associated constraints were relaxed. The user's input is needed to select the constraints to be 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 **distorted** solutions to the original problem.

To facilitate user input, preferences are assigned to the constraints of the problem reflecting *a priori* a hierarchy among the constraints, but these preferences are not mandatory in our system. A metric is derived from the preferences and measures the quality. This metric allows an automation of our solver to find all the distorted solutions sorted by quality.

The user-driven version is of great interest when *a priori* preferences are hard to determine (this is often the case!). The user can restrict interactively the search to a subset of distorted solutions. Explanation-based constraint programming gives a complete control to the user: This is important in such an *intellectual* activity.

5.3 Application to the Composite design pattern

We model the Composite design pattern by mean of a CSP (see Example 4). The input source code, presented in Example 2, is then modelled: The classes and the relationships among classes are encoded into tables (see Example 5).

> **Example 4 (Modelling the Composite design pattern) :**
> The Composite design pattern, as presented in Example 1, is modelled by associating a variable with each class defined (`Component`, `Composite` and `Leaf`) and by constraining the values of these variables according to the relationships among classes: $\mathtt{composite} < \mathtt{component}$, $\mathtt{leaf} < \mathtt{component}$, and $\mathtt{composite} \supset \mathtt{component}$.

Our explanation-based constraint solver, PaLM [Jussien and Barichard, 2000], solves this CSP to identify subsets of classes whose structures are similar to the micro-architecture of the design pattern by giving a set of distorted solutions (see Example 6).

The source code is then modified accordingly, leading to the corrected kernel of our application presented in Example 2, on the right.

6 First results

We developed a tool, Patterns Trace Identification, Detection and Enhancement for Java[4] (Ptidej see

[4] A demonstration is available at:
`www.yann-gael.gueheneuc.net/Work/PtidejDemo.html`

Example 5 (Modelling the source code) :
The source code of the application, Example 2, involves seven classes: `AbstractDocument`, `Element`, `Title`, `Paragraph`, `ParaIndent`, `Document`, and `Main`. The domain of each variable of the CSP presented Example 4, `component`, `composite`, and `leaf`, is of size 7 (one slot for each possible class from the source code). We define a generic model to encode classes from the source code in our system. This generic model is a table:

```
PClass
    name:           string,
    superclasses:   list[PClass],
    components:     list[PClass],
    componentsType: PClass,
    relatesTo:      list[PClass],
    doNotRelateTo:  list[PClass]
```

The relationships among classes are encoded in this model and are used to check the relationships required by the design pattern:

- `name` represents the name of the class.
- `superclasses` is the list of the direct superclasses of the class represented by this ephemeral object.
- `components` is the list of all the components aggregated by the class represented by this ephemeral object. `componentsType` is the common super-class of all the components.
- `relatesTo` is the list of all the classes that are known by the class represented by this ephemeral object.
- `doNotRelateTo` is the list of all the classes that are unknown to the class represented by this ephemeral object.

We can deduce automatically all the needed information from the source code of the application.

Example 6 (Solutions) :
The resolution of the CSP modelling the Composite design pattern on the application to produce representations of documents (see Figure 2, on the left) produces results of the form:

- $< dist.sol.$#>.<Quality>.component = <a class>
- $< dist.sol.$#>.<Quality>.composite = <a class>
- $< dist.sol.$#>.<Quality>.leaf = <a class>

A solution, without constraint `Component` $<$ `Composite`, of weight 50, is:

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

There are five other solutions. The solutions are automatically provided by our tool.

its interface on Figure 3). This tool performs the different steps presented in Section 2.2 to improve the quality of a source code from an architectural point of view. This tool, written in JAVA, accepts JAVA source code. The solver is written in CLAIRE [Caseau and Laburthe, 1996] using the PALM explanation-based constraint solver [Jussien and Barichard, 2000] developed on top of the CHOCO constraints system [Laburthe, 2000].

It allows:

- To load and to visualize (using an OMT-like notation) an application written in JAVA
- To generate a model of the application for the constraint system
- To call the explanation-based constraint system PALM on this model to detect the referenced design patterns
- To visualize the (real and distorted) solutions found
- To perform the needed transformations on the source code to make it similar to a design pattern and thus to improve its quality
- And to load and to visualize the modified application

Three design patterns are referenced by the tool: The Composite design pattern presented in Section 2.1; the Facade design pattern, that models relationships between a set of *client* classes and a set of classes forming a *sub-system* through a unique `Facade` class with no mutual knowledge (see Example 7); and the Mediator design pattern, a design pattern similar to Facade in which the clients classes and the classes of the sub-system may know about one another.

Example 7 (An overview of the Facade design pattern) :
The general structure of the Facade design pattern is shown Figure 4. The Facade design pattern is composed of three classes: `Clients`, `Facade`, and `SubsystemClasses`, such as: `clients` $\triangleright$ `facade` $\triangleright$ `subsystemClasses`, `subsystemClasses` $\not\triangleright$ `facade` $\not\triangleright$ `clients`, and `subsystemClasses` $\not\triangleright$ `clients`. The `clients` and `subsystemClasses` variables, encoding sets, are not enumerated, letting the propagation system to remove unfeasible solutions.

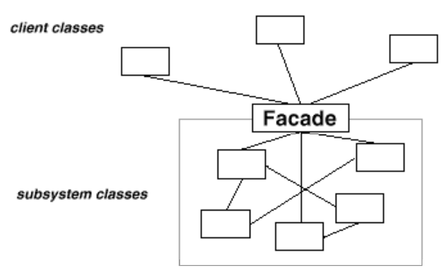

Figure 4: The Facade design pattern

Any design pattern may be modelled and referenced by our tool. However, the structural design patterns (like Compos-

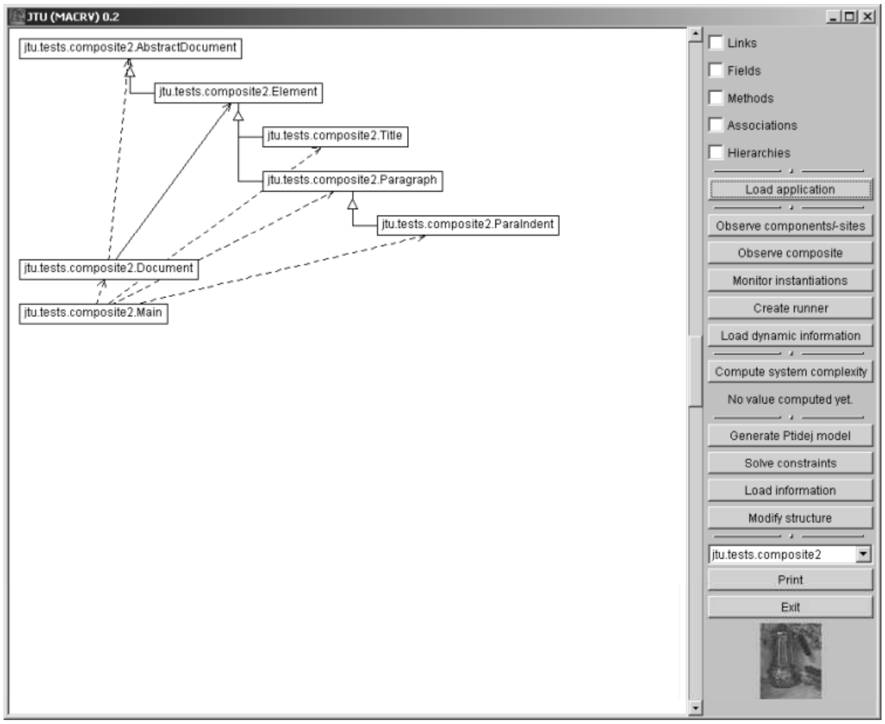

Figure 3: Interface of PTIDEJ.

ite or Facade) are easier to model than the behavioral or creational design patterns because these latter need statically undecidable information (such as the type of a particular object in a generic collection). The modelling of the Abstract Factory, Observer and Singleton design patterns is in progress. An Abstract Factory provides an interface to build families of related objects without specifying their concrete classes. The behavioral Observer design pattern defines dependencies among objects such that all dependent objects are notified and updated when one of the related objects changes. The creational Singleton design pattern ensures that a class has a unique instance in a system, and provides a global entry point for it.

We applied our tool on different systems. In particular, we applied our approach on two packages of the JAVA class libraries: The `java.awt` and `java.net` packages. All well-known occurrences of the Composite and Facade design patterns have been identified, as well as other less-known distorted occurrences. These results are promising but we need to analyze manually the packages to check that all possible distorted solutions have been identified using our models: We need to check the *modelling* of the design patterns not the *method*, which has been proven to be complete.

7 Conclusion

In this article, we presented an original use of explanation-based constraint programming to propose a solution to a difficult problem: The identification of design patterns in object-oriented source code. Explanations are used to provide a user-friendly system: Distorted design patterns are identified and explained, the search can be completely driven by the user, *etc.*

We developed a library of dedicated constraints to solve this problem. These constraints are used in our tool, PTIDEJ. The first results of our approach are satisfying because they allow to propose, for the first time, a tool to solve this problem.

Our current work concerns the definition of more relationships among classes, the extension of the library of constraints and the application of the constraints to other systems, such as JHOTDRAW [Gamma, 1998], and of course PTIDEJ itself!

Acknowledgements

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

References

[Bayardo Jr. and Miranker, 1996] Roberto J. Bayardo Jr. and Daniel P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *AAAI'96*, 1996.

[Bessière, 1991] Christian Bessière. Arc consistency in dynamic constraint satisfaction problems. In *Proceedings AAAI'91*, 1991.

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

[Caseau and Laburthe, 1996] Yves Caseau and François Laburthe. CLAIRE: Combining objects and rules for problem solving. *Proceedings of JICSLP, workshop on multi-paradigm logic programming*, 1996.

[Demeyer *et al.*, 2000] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Object-oriented reengineering OOPSLA'00 tutorial. *OOPSLA Tutorial Notes*, 2000.

[Fowler, 1999] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fuchs *et al.*, 2000] B. Fuchs, J. Lieber, A. Mille, and A. Napoli. An Algorithm for Adaptation in Case-Based Reasoning. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000), Berlin, Germany*, pages 45–49, 2000.

[Gamma *et al.*, 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[Gamma, 1998] Erich Gamma. JHotDraw, 1998. Available at http://members.pingnet.ch/gamma/JHD-5.1.zip.

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

[Guéret *et al.*, 2000] 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, 2000.

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

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

[Jussien and Barichard, 2000] Narendra Jussien and Vincent 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, September 2000.

[Jussien and Boizumault, 1997] 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.

[Jussien and Lhomme, 2000] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Seventh National Conference on Artificial Intelligence – AAAI'2000*, pages 169–174, Austin, TX, USA, August 2000.

[Jussien *et al.*, 2000] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[Krämer and Prechelt, 1996] Christian Krämer and Lutz Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. *Proceedings of the Working Conference on Reverse Engineering*, pages 208–215, 1996.

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

[Mancoridis *et al.*, 1998] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. *Proceedings of ICSM*, 1998.

[Régin, 1995] Jean-Charles Régin. *Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique*. Thèse de doctorat, Université de Montpellier II, 21 December 1995. In French.

[Richner and Ducasse, 1999] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. *Proceedings of ICSM*, 1999.

[Rumbaugh *et al.*, 1991] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[Schiex and Verfaillie, 1994] Thomas Schiex and Gérard Verfaillie. Nogood Recording fot Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

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