This paper has been accepted at the ECOOP 2001 workshop for PhD Students in Object-Oriented Systems.

Design Patterns: A Round-trip

Hervé Albin-Amiot*, Yann-Gaël Guéhéneuc†
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
44307 Nantes Cedex 3
France
{albin|guehene}@emn.fr

Abstract

Design patterns are of major interest to increase software quality and abstraction level. However, design patterns are difficult to choose, to apply, and to recover. We propose a set of tools to use design patterns in a round-trip fashion. We define a meta-model to describe design patterns. This meta-model is specifically oriented towards design patterns instantiation and detection. We develop a source-to-source transformation engine to modify the source code to comply with design patterns descriptions. Meanwhile, we use an explanation-based constraint solver to detect design patterns in source code from their descriptions. With these tools, we hope to offer a mean to apply and to recover design patterns without overhead for the developers.

1 Problems

Developing quality code is a major concern for the software community. Producing bug-free, extensible, and adaptable [Tok99] code is a hard task that requires skills, experience, and a deep understanding of the structure and behavior [RD99] of the software under development and of its context.

The first problem encountered when developing quality code is in understanding the legacy of previous developments. As Enzo Torresi put it, in the Bits and Bytes column of the *San Jose Mercury News* (November the 15th, 1990; cited in [Pot90]):

> *The reason God was able to create the world in seven days is that he did not have to worry about the installed base.*

An experienced programmer can often reconstruct much of the hierarchy of a program's design by recognizing commonly used data structures and algorithms, and by knowing how they typically implement high-level abstractions [RW90]. However,

*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.

tools do not yet support automatic creation of these high-level abstractions based on existing systems [PM99].

The second problem encountered when developing quality code is in building a new piece of software with the best available methodologies and techniques, and in integrating this new piece of software within the legacy code. An experienced programmer often follows idioms to solve low-level code-related problems, and possesses a set of solutions to recurring high-level architectural problems. However, tools do not yet support the description and the automatic application of these high-level solutions on under-development or existing systems.

These two problems are related to a global process of software design known as *round-trip*. As defined by Booch [Boo91], p. 517 (emphasis ours), round-trip is:

> *A style of design that emphasizes the incremental and iterative development of a system, through the refinement of different yet consistent logical and physical views of the system as a whole; The process of object-oriented design is guided by the concepts of round-trip gestalt design;* Round-trip gestalt design is a recognition of that fact that the big picture of a design affects its details, and that the details often affect the big picture.

Finally, the third problem arising when developing quality code is in forging a common vocabulary among developers. This vocabulary is composed of recurring problems and their solutions, of design and coding choices, and of shared experiences. This vocabulary eases the development and facilitates the relationships and the recognition among developers [Cop99].

2 Solution

As many other authors, we advocate the use of design patterns to address the three previous problems.

The ultimate goal of design patterns is to help in designing software in a new way and with better quality [Sou95]. Already, design patterns are being used as building blocks for software architecture [TC97]. Figure 1 shows frameworks, kits and applications development cycles: These three technologies benefit from the use of design patterns.

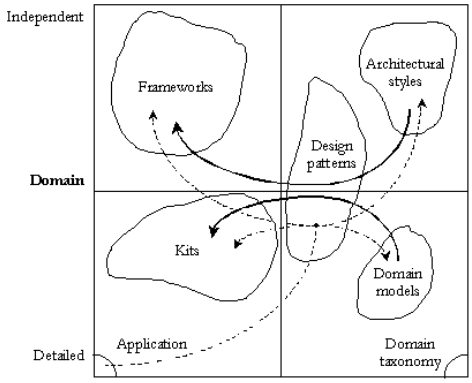

Figure 1: *This topology, borrowed from [TC97], shows the path for developing a new framework (bold upper arch) and the path for developing a new kit (bold lower arch). Both frameworks and kits benefit from design pattern guidance, even though they start from different sectors of the topology. Once design patterns are identified in the application, the dotted arrows show that different sectors of the topology can be reached using this knowledge about design patterns.*

However, the application of design patterns is error-prone [PUT+00], and the detection in legacy code of design pattern is difficult [Ban98].

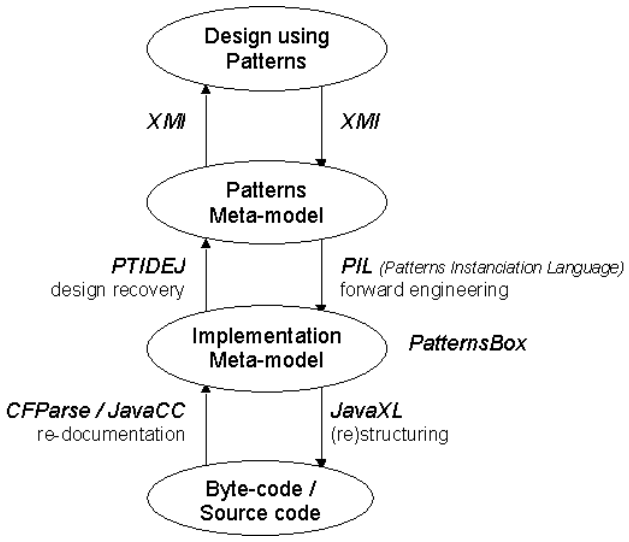

Figure 2: *A taxonomy of software engineering terms and of our tools. Ovals represent abstraction levels related to designing software with design patterns. Between each abstraction levels, arrows indicate software engineering processes and techniques or tools that support these processes. The* forward *and* reverse *arrows illustrates the round-trip between abstract levels.*

We propose a set of methods and tools to create, to apply, and to detect design patterns. Figure 2 shows an overview of our research. There are still some missing pieces, this article presents the current state of our research.

These methods and tools use proven technologies: In section 3, we introduce a meta-model specifically oriented towards application and detection of design patterns. In section 4, we present our source-to-source transformation engine. This transformation engine is driven by the meta-model to apply automatically design patterns on source code. In section 5, we propose the use of an explanation-based constraint solver to detect automatically design patterns described using our meta-model. The methods and tools we propose cover the entire life-cycle of a software and address the problem of round-trip. Finally, in section 6, we give pointers to future directions of research.

Appendix A presents a succinct example of the previous technologies and tools on a simple application. Appendix B gives the declaration of the Composite pattern using our meta-model.

3 A Meta-Model (Creation)

Our meta-model provides a way to describe the structural and behavioral aspects of design patterns (such as those proposed in [GHJV94]).

There exist several meta-models for representing design patterns but none are specifically designed towards detection and code transformation. [RF00] and [PW96] introduces meta-models for design patterns instantiation and validation but without support for code generation. In the tool PatternGen [SGJ00], the meta-model does not support code generation (another

module handles it) and it offers no patterns detection. In [FMW97], the fragments-based system allows only representation and composition of design patterns.

Our meta-model is inspired from the work of [PW96]. We did not define it as an extension of an existing meta-model (such as UML). We built it from scratch using the JAVA programming language and following the JAVABEANS *formalism* (properties, idioms, and introspection). *The meta-model handles uniformly the instantiation and the detection of design patterns.* It embodies a set of entities[1] and the interaction rules among them. An abstract model of a design pattern represents the design pattern *generic* micro-architecture and behavior. An abstract model of a pattern is expressed using constituents of the meta-model. The abstract model of a pattern contains the design pattern micro-architecture [AFC98] and behavior. An abstract model of a pattern may be reified: The design pattern abstract model is cloned and adapted to the current context (names, cardinalities, specificities of the code, ...) and becomes a concrete model: A first-class object that we can manipulate and about which we can reason. We use reified design patterns to generate source code and to detect sets of entities with similar architecture and behavior in source code. We instantiate and detect the design patterns according to their abstract models, rather than according to external specifications.

The detailed rules of operation and use of the meta-model are out of the scope of this paper and will not be further discussed.

Our meta-model suffers some limitations: (1) It lacks expressiveness with respect to the most dynamic aspect of the application because it expresses relationships among entities, not among instances; (2) It needs to be further specialized to allow a finer-grain control over the constraints and the transformation rules.

[1] In JAVA, an entity may be either a class, an abstract class, or an interface.

4 A Source-to-Source transformation engine (Application)

From the concrete and abstract models of a design pattern, we deduce the transformation needed to integrate this design pattern in the source code.

A concrete model of a design pattern is a reified abstract model. The concrete model is different from its abstract model: Entities of the micro-architecture are named, and, to a certain extent, the overall design pattern micro-architecture and behavior are different from the abstract model. For example, an abstract model may specify that a participant of its micro-architecture is an interface. In the concrete model, this participant may be an abstract class: The difference between an interface and an abstract class may be small enough to be accepted. Differences between concrete and abstract models are defined and enforced by the abstract model. The differences are tolerated by the abstract model as long as the *principle* of the design pattern is not compromised.

From the differences between an abstract model and a concrete model, we deduce the transformations needed to transform the source code such that it complies with the abstract model specification. For example, if the abstract model requires one participant to inherit from another participant, and the concrete model contains the two participants but without inheritance link, then the adequate transformation is to add an inheritance link between the two participants.

We express these transformation as source-to-source transformations using

JavaXL, our source-to-source transformation engine. JavaXL differs from work such like [TC98] or [Chi98] because *our major concern is to modify as little as possible the source file.* When a user applies a design pattern on her source code, JavaXL performs only the few needed changes. Thus, after applying a design pattern, the user may continue her development without any loss of comments, layout, or specific coding conventions.

The major problem we face is the lack of formalism in deducing transformations from a concrete model according to its abstract model. Right now, the transformations are included into the abstract model by hand. We lack a tool and the underlying techniques to go automatically from an abstract model to the source-to-source transformations and then to transform the source code.

5 An Explanations-Based Constraints Solver (Detection)

We deduce a constraint satisfaction problem (CSP) from an abstract model of a design pattern. This CSP represents the problem our explanation-based constraint solver, PaLM [JB00], solves to identify, in a given set of classes, a sub-set of classes which structure is identical or similar to the micro-architecture defined by the design pattern abstract model (the problem of identifying design patterns is similar to the problem of searching sub-graphs in a graph and thus is NP-complete). The solutions of the CSP are concrete models: Instances of the corresponding design pattern abstract model.

The problems of detecting design patterns in source code to help documenting or understanding legacy systems is subject of many works [Bro96, KP96, Wuy98, MMCG98, RD99]. Among those studies, the use of logic programming is the technique with the greatest interest [Wuy98]: A design pattern is described as a set of 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 architectures of a design pattern. The rules must be extended to introduce the missing distorted cases and to obtain more solutions. Consequently, the rules become quickly impossible to manage. The addition of new distorted solution requires thinking about all the possible distortions when conceiving the system of rules.

We propose the use of explanations-based constraint programming to alleviate the limitations of logic programming.

First, non-distorted – complete – solutions are computed. This computation ends by a contradiction (there is no more complete solution). Explanation-based constraint programming provides a contradiction explanation for this failure: The set of constraints justifying that other combinations of entities do not verify the constraints describing the wanted design pattern. We do not need to relax other constraints than the constraints provided by the contradiction explanation: We would find no other distorted solutions. Explanation contradiction provides insights on what distorted solutions are available. *This information about possible distorted solutions allows the user to lead the search towards interesting distorted solution (from her point of view) by pointing out constraint to relax.* Removing a constraint suggested by the contradiction explanation does not necessarily lead to new distorted solutions

but the process is applied recursively.

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

The set of constraints to detect a given design pattern is so far written by hand according to the abstract model of the pattern. It is one of our objectives to generate automatically the set of constraints needed to detect a design pattern from the abstract model of this design pattern. Furthermore, we want to use the weights associated with each constraints as a mean to differentiate distorted solutions. However, setting weights automatically may not be possible and may impede further the automation.

6 Future work

We plan to further investigate the possibilities of including domain-specific knowledge and mechanisms to measure the degree of improvement – using design metrics. Another direction of research consists in tightly integrating constraints and transformation rules in the meta-model. It will imply a finer-grain definition of the meta-model that may impede its robustness and conciseness. Two distinct meta-models may be a solution: One (high-level) for the specification of design patterns; One (low-level) for the descriptions of the source code, with the constraints and the transformation rules linking the two levels. We will also test our tools on larger real-life applications to evaluate precisely their efficiency and scalability. For example, JHotDraw [Gam98] contains more than 125 classes and identifies several design patterns.

A Example of the prototype

First, we declare the abstract model of the Composite pattern using the constituents of the meta-model. Then, we generate the constraints used to detect groups of entities similar to the Composite pattern into a given source code modeled with the same meta-model.

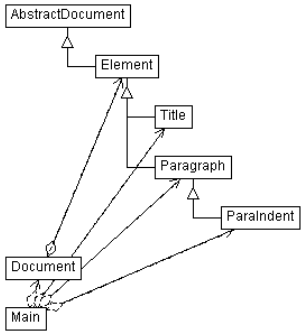

Figure 3: *The architecture of the example application. This application produces representations of documents.*

The input source code in which we want to detect the Composite pattern is a simple application of text documents description, see in Figure 3 for a graphical UML-like representation. A `Document` contains elements (class `Element`), which can be `Title`, `Paragraph` or indented paragraph, `ParaIndent`. The `Main` class creates an instance of `Document` and fills it up with titles, paragraphs, and indented paragraphs. The relationships among classes `Element`, `Document`, `Title`, `Paragraph` and `ParaIndent` are typical of a Composite pattern. However, class `Document` should be subclass of `Element`, because we want a uniform interface between the composition of objects and individual objects.

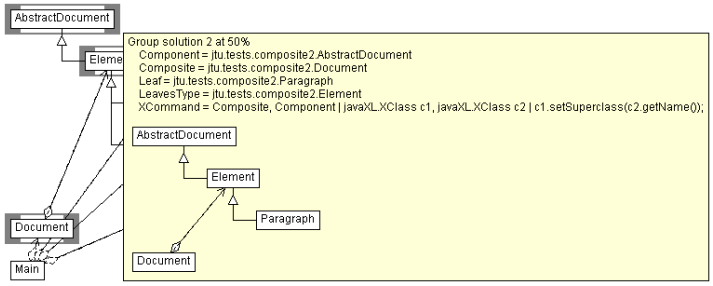

Figure 4: *A suggestion of modification. The tool suggests that the class* `Document` *be made sub-class of* `Element` *to conform to the micro-architecture advocated by the* Composite *pattern.*

We apply the constraints defined for the Composite pattern on the source code corresponding to the application modelled using the same meta-model. The constraint solver generates the set of all the groups of entities similar to the Composite pattern abstract model. These groups are visible on Figure 4 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 (`XCommand` field). Figure 4 shows these pieces of information for a particular group.

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());
```

The class playing the role of `Composite` must be subclass of the class playing the role of `Component`. In our example, the class `Document` must be subclass of class `Element`. The transformation engine performs automatically the modifications on the application by executing the `XCommand` on the source code. Figure 5 illustrates the resulting architecture of the application.

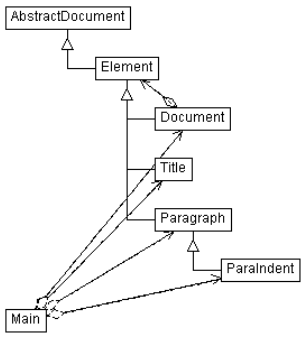

Figure 5: *The modified architecture of the application as suggested by the* Composite *pattern: Class* `Document` *is now a sub-class of class* `Element`.

B Example of the Composite pattern abstract model

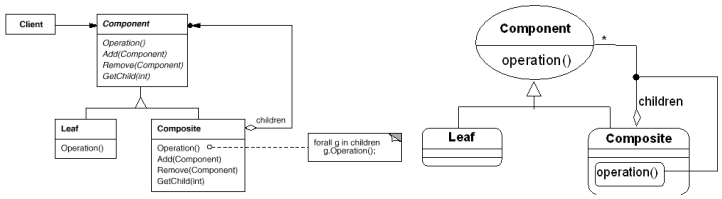

Figure 6: (On the left) *The UML-like diagram of the* Composite *pattern, from [GHJV94].*

Figure 7: (On the right) *A representation of the* Composite *pattern abstract model. Rounded squares represent instances of* `PClass`*; ovals, instances of* `PInterface`*; and boxed method, instances of* `PDelegetingMethod`*. Instances of* `PAssoc` *and realization links use the* UML *notation.* `PClass`*,* `PInterface`*,* `PDelegetingMethod` *and* `PAssoc` *are elements of our meta-model, not described here.*

Composite pattern abstract model definition
class Composite extends Pattern {
Declaration takes place in the Composite class constructor
Composite(. . .) { . . .
Declaration of the "Component" actor
component = new PInterface("Component") operation = new Pmethod("operation") component.addPElement(operation) this.addPEntity(component)
Declaration of the association "children" targeting "Component" actor with cardinality n
children = new PAssoc("children", component, n)
Declaration of the "Component" actor
composite = new PClass("Composite") composite.addShouldImplement(component) composite.addPElement(children)
The method "operation" defined into "Composite" actor implements the method operation of "Component" actor and is linked to its through the association "children"
aMethod = new PdelegatingMethod("operation", children) aMethod.attachTo(operation) composite.addPElement(aMethod) this.addPEntity(composite)
Declaration of the "Leaf" actor
leaf = new PClass("Leaf") leaf.addShouldImplement(component) leaf.assumeAllInterfaces() this.addPEntity(leaf) }
Declaration of specific services dedicated to the Composite pattern
. . .
For example, the service "addLeaf" to dynamically adds "Leaf" actor to the current instance of the Composite pattern

void addLeaf(String leafName) {
PClass newPClass = new PClass(leafName)
newPClass.addShouldImplement((PInterface)getActor("Component"))
newPClass.assumeAllInterfaces()
newPClass.setName(leafName)
this.addPEntity(newPCclass)
}

References

[AFC98] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. *Proceedings of the 6 th Workshop on Program Comprehension*, pages 153–160, 1998.

[Ban98] Jagdish Bansiya. Automating design-pattern identification. *Dr. Dobb's Journal*, June 1998.

[Boo91] Grady Booch. *Object-Oriented Design with Applications*. Number 0-805-30091-0. The Benjamin/Cummings Publishing Company, Inc., 390 Bridge Parkway - Redwood City, California 94005 - USA, 1991.

[Bro96] Kyle Brown. Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, University of Illinois at Urbana-Champaign, 1996.

[Chi98] Shigeru Chiba. Javassist a reflection-based programming wizard for Java. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98*, 1998.

[Cop99] James O. Coplien. Reevaluating the architectural metaphor: Toward piecemeal growth. *IEEE Software*, 16(5):40–44, September/October 1999.

[FMW97] Gert Florijn, Marco Meijers, and Pieter Van Winsen. Tool support for object-oriented patterns. *Proceedings of ECOOP*, 1997.

[Gam98] Erich Gamma. JHotDraw, 1998. Available at http://members.pingnet.ch/gamma/JHD-5.1.zip.

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

[JB00] 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.

[KP96] 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.

[MMCG98] 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.

[PM99] Louis Perrochon and Walter Mann. Inferred designs. *IEEE Software*, 16(5):46–51, September/October 1999.

[Pot90] Kathleen Potosnak. Human factors – pruning your programs' unused functions. *IEEE Software*, 7(1):122–124, January 1990.

[PUT+00] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 2000.

[PW96] Bernd-Uwe Pagel and Mario Winter. Towards pattern-based tools. *Proceedings of EuropLop*, 1996.

[RD99] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. *Proceedings of ICSM*, 1999.

[RF00] Pascal Rapicault and Mireille Fornarino. Instanciation et vérification de patterns de conception : Un méta-protocole. *Proceedings of LMO, in French*, pages 43–58, 2000.

[RW90] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[SGJ00] Gerson Sunyé, Alain Le Guennec, and Jean-Marc Jézéquel. Design patterns application in UML. *Proceedings of ECOOP*, 2000.

[Sou95] Jiri Soukup. *Implementing Patterns*, chapter 20. Addison-Wesley, 1995.

[TC97] William Tepfenhart and James Cusick. A unified object topology. *IEEE Software Special Issue on Objects, Patterns, and Architectures*, 14(1):31–35, January 1997.

[TC98] Michiaki Tatsubori and Shigeru Chiba. Programming support of design patterns with compile-time reflection. *Proceedings of the Workshop on Reflective Programming in C++ and Java at OOPSLA'98, ISSN 1344-3135, Vancouver, Canada*, pages 56–60, October 1998.

[Tok99] Lance Aiji Tokuda. *Evolving Object-Oriented Designs with Refactorings*. PhD thesis, University of Texas at Austin, December 1999.

[Wuy98] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. *Proceedings of TOOLS USA*, pages 112–124, 1998.