Towards Better Support for Pattern-Oriented Software Development

Dietrich Travkin

Software Engineering Research Group,
Heinz Nixdorf Institute & Department of Computer Science,
University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany,
travkin@upb.de

Abstract. Design patterns document approved solutions for recurring design problems. Due to their vague description such patterns are widely applicable, but their application is error-prone. Since pattern applications are rarely documented, the originally intended design tends to deviate during software evolution. With my research I aim at explicitly modeling and validating pattern applications in design models in order to maintain an up-to-date documentation and reduce design deviation.

1 Introduction

When developing or adapting software, developers have to solve similar design problems over and over again. In order to reuse good solutions for common design problems, experts have documented approved solutions as *software patterns*. One of the most famous summaries of software patterns are *Design Patterns* described by Gamma et al. [7]. A design pattern has a unique name and consists of a problem description, the intent, a general description of the design that solves the problem, and consequences.

For the purpose of making such solutions reusable in many cases, they are described informally and in a very general way. Instead of documenting a concrete design, only the main idea is described. Unfortunately, this results in a lot of work and complicated decisions that are necessary to transfer a solution to and refine it for a concrete software design. The same holds for other software pattern descriptions, too [3, 2, 10, 8, 1].

Any pattern's exemplary design described in literature has to be manually adapted to the existing software design, i.e. a developer has to identify design parts that are to be added or adapted. The resulting design changes are usually performed manually. Furthermore, once a software pattern is applied, it is rarely documented or the current design deviates from its documentation due to subsequent design changes. This may result in design erosion [12] and the loss of the original developer's intent (e.g. to decouple certain software parts, make them replaceable and thus make the software more flexible).

My research aims at the development of methods and tools for model-driven software development that support software engineers in formally specifying software patterns, flexibly applying such patterns to an existing software design model, documenting the pattern applications, and validating the design in case of changes affecting the pattern applications.

2 Related Work

Since the introduction of software patterns, esp. design patterns [7], about 15 years ago, several researchers tried to formally specify patterns [11], e.g. in order to develop tools that automatically or interactively apply a pattern to an existing design or check whether the design conforms to a pattern. Most of them use sets of logic formulas to describe a pattern. But in many cases formulas are less comprehensible dable than graphical design description languages like UML, where relations are clearly visualized.

An example for a graphical and yet formal pattern specification language is the LePUS language [9, 11]. The proposed approach enables developers to check if a pattern implemention complies with its formal LePUS specification. A major drawback is the need for extensive user interaction before a conformance check can be performed.

There are also some approaches where patterns are specified in UML or UML-like languages. One of them uses specialized class and sequence diagrams as well as OCL [6, 11]. However, currently there is no tool support for validation of applied patterns.

Furthermore, none of the approaches mentioned so far [11] provides support for application of patterns in an existing design and applied patterns cannot be documented in design models.

An early research result [5] enables developers to specify their own patterns, to apply patterns in an existing Smalltalk application, and to check previously specified constraints in order to ensure that the implementation complies with the pattern. Nevertheless, the pattern application and validation operations have to be implemented manually, which complicates the specification of new patterns. Furthermore, they are applied to code, not to design models which are preferred in a model-driven software development process.

Commercial tools like IBM Rational Software Architect[1], Sparx Enterprise Architect[2], and Borland Together [3]also use UML to specify software patterns. In the Rational Software Architect, pattern applications are explicitly modeled and thus documented in UML models. Applying a pattern modifies a UML model and creates classes, methods, etc. But for each new pattern an expert has to manually implement the pattern application operations. Enterprise Architect and Together only save exemplary design parts in order to re-instantiate them later. Enterpise Architect in addition provides rudimentary functionality

[1] http://www-01.ibm.com/software/awdtools/swarchitect/websphere
[2] http://www.sparxsystems.de
[3] http://www.borland.com/us/products/together/index.html

to merge such an example model with an existing design. Besides, neither of these tools supports validation of applied patterns.

3 Proposed Approach

With my research I focus on model-driven software development. I assume that there is a design model that is created by a developer and used for code generation. Software patterns will be applied on such a model and will be documented and checked in such a model.

In order to formally specify software patterns, I propose a language that is capable of describing the pattern structure and most of the corresponding behavior as well as constraints that define the properties to be preserved after a pattern application. In order to base my language on concepts such as types, attributes, operations, type relations like associations and inheritance, as well as behavioral aspects like read or write access, delegation, etc., I restrict the design models to object-oriented models like the UML. In addition, I plan to support compositions of pattern specifications in order to increase reuse and to simplify the specification of complex patterns.

I am currently developing a meta-model and a graphical syntax for the pattern specification language that defines all language constructs. A first draft of a pattern specification based on this language is illustrated in Figure 1.

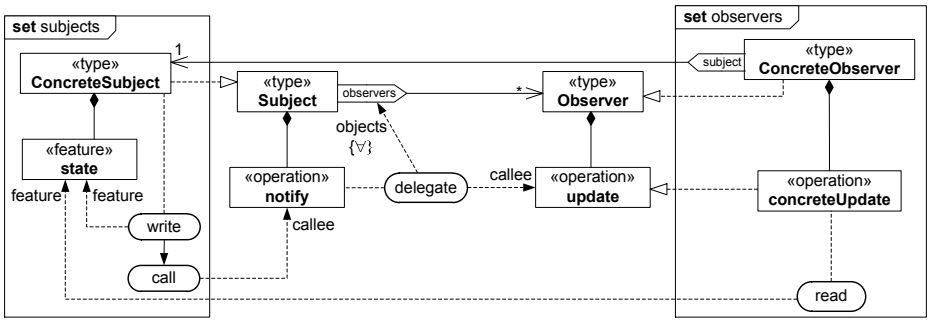

Fig. 1. Sketch of an Observer pattern specification

As an example, I specified the Observer pattern [7]. Between the types *Subject* and *Observer* a reference *observers* with cardinality * is specified. The subject's *notify* operation delegates its functionality to an observer's *update* operation, which is illustrated by a *delegate* node and corresponding arrows. In contrast to other approaches, it is also specified on which objects the *update* operation is to be called: in this case on each *Observer* object (denoted as {∀}) referenced by the *observers* reference. Moreover, control flow can be specified as well. For example, a write access to the *state* feature (this can be an attribute or a reference) is followed by a call of the *notify* operation (denoted as solid arrow

between the nodes *write* and *call*). Set fragments (rectangles labelled with *set*) specify groups of elements that as a whole can be created multiple times in the target design model, e.g. a concrete observer type with a corresponding concrete update operation can be created several times.

I plan to provide tool support for the specification of patterns as well as their semi-automated application. In order to apply a pattern in an existing design model (e.g. a model of classes), a developer first specifies which parts of the design he would like to reuse. For that purpose, the developer maps existing design model elements (e.g. existing classes) to the pattern roles (nodes in Figure 1) that are to be played. In a next step, the tool will automatically adapt the design model by creating new model elements for each role that is not mapped. This way, the pattern application remains flexible and the developer decides in each situation what is going to be re-used or created anew.

The actual pattern application will be performed by means of a model transformation which is generated based on the selected pattern and the selected design model elements to be re-used. Due to their formal semantics, available tool support, and our research group's expertise, I plan to use so-called *story diagrams* [4] to specify the transformations. Story diagrams are specialized UML activity diagrams where the actions are graphically specified as graph grammar rules. These rules describe the object structures to be found and corresponding structure modifications.

Pattern applications will be documented as first-class constructs in the target modeling language so that developers do not loose track of applied patterns. This can be done by annotating the original design model (e.g. a class diagram) and visualizing the elements involved in a pattern application as well as the roles they are playing. For this purpose, I will provide an adequate notation and annotation mechanism.

If the design model is changed after a pattern application, the affected design elements are to be checked if they still comply with the pattern specification. For example, it can be checked if once created model elements still exist and if the specified constraints are satisfied, e.g. the design complies to specified access restrictions.

4 Summary and Evaluation Idea

Tool support as described above can significantly ease software development. By applying patterns, design solutions can be re-used on a high level of abstraction, the pattern applications are automatically documented in the model, and corresponding constraints are automatically checked after design modifications so that the original intent is preserved. The major goals of this approach are improved comprehendability of design models and reduced design deviation.

I plan to provide a prototype and to apply my development methodology to a realistically-sized software project in order to assess the benefits and drawbacks of my approach. Furthermore, I plan to compare traditional and the proposed pattern-oriented software development in an experimental setting.

References

1. Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns – Best Practices and Design Strategies*. Core Design Series. Prentice Hall, Sun Microsystems Press, 2 edition, 2003.
2. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*, volume 1 of *Software Design Patterns*. John Wiley and Sons, Ltd, 1996.
3. James O. Coplien and Douglas C. Schmidt, editors. *Pattern Languages of Program Design*, volume 1 of *Software Patterns Series*. Addison-Wesley, 1995.
4. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science*, pages 296–309. Springer, 1998.
5. Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool Support for Object-Oriented Patterns. In *ECOOP'97 – Object-Oriented Programming*, volume 1241/1997 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997.
6. Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, 30(3):193–206, 2004.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.
8. Robert S. Hanmer. *Patterns for Fault Tolerant Software*. Software Design Patterns. John Wiley and Sons, Ltd, 2007.
9. Jonathan Nicholson, Epameinondas Gasparis, Amnon H. Eden, and Rick Kazman. Automated Verification of Design Patterns with LePUS3. In *1st NASA Formal Methods Symposium, Moffett Field, California*, April 2009.
10. Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2 of *Software Design Patterns*. John Wiley and Sons, Ltd, 2000.
11. Toufik Taibi, editor. *Design Pattern Formalization Techniques*. IGI Publishing, Hershey, PA, USA, 2007.
12. Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *The Journal of Systems and Software*, 61(2):105–119, 2002.