

Representing Design Patterns as Design Components

Mária SMOLÁROVÁ, Pavol NÁVRAT

Department of Computer Science and Engineering, Slovak University of Technology,
Ilkovičova 3, 812 19 Bratislava, Slovakia
smolarova@dcs.elf.stuba.sk, navrat@elf.stuba.sk

Abstract. Patterns should be incorporated in a software system in an explicit form so that they will be known to subsequent developers. In order to support pattern application, two basic requirements should be fulfilled: traceability and visibility. In our method, design patterns are represented as design components that may be placed in the library of pattern prototypes. From a prototype, an instance can be created when designing an actual application. In the paper, we describe prototype structure and the way of deriving pattern instances from the prototype.

1 Introduction

Design patterns are becoming an accepted way of communicating design knowledge among software developers. With current catalogues of design patterns, software developer can choose a pattern and apply it to his/her actual design. However, applying design patterns may not be easy. Designs patterns, while useful and important, often complicate design and may bring for example additional levels of indirection into actual design [6]. Application of design patterns may also be tedious because of a lot of manual work to be done [3]. Beside that, it is still not obvious how pattern application influences the software development process. One can imagine the case when patterns come first and the application specific code is incorporated into the skeleton given in the particular pattern. But it is also possible to apply patterns when (a part of) an application exists. In this case, due to the pattern presence it may be necessary either to refactor existing parts or align new ones or both. In both cases, patterns should be incorporated in a software system in an explicit form so that they will be known for subsequent developers. In order to support pattern application, two basic requirements should be fulfilled:

1. pattern traceability: patterns intended as an unit at one development stage should be traceable to next development stages
2. pattern visibility: after pattern application, the parts related to patterns are visible.

Current way of formulating design patterns seems to be a satisfying way of formulation. But for a user who wants to apply patterns to his/her design problem and hopes also for some tool support, the current pattern presentation is not satisfactory. When a pattern from a catalogue is chosen and manually applied, i.e. incorporated into the actual design, neither visibility nor traceability of design patterns can be guaranteed to be preserved any more. In order not to lose an information about pattern presence in the actual design, a more appropriate representation of patterns is needed. We suppose that design patterns are represented as design components that may be placed in the library of pattern prototypes. From a prototype, an instance may be created when an actual application is designed.

This paper is organised as follows: Section 1 starts with motivation, introduces new terminology and describes shortly the Bridge pattern. Our technique for representing design patterns is based on pattern prototypes. Section 2 describes prototype structure in general and as example gives the Bridge pattern prototype. The way in which pattern instances are derived from the prototype is shown in Section 3.

1.1 Motivation

A pattern gets easily lost after it is applied because design patterns do not have direct counterparts in the design or implementation phase. An applied pattern is spread over several places in the design and code. To record pattern presence seems to be substantial from a perspective of subsequent developments.

In any nontrivial software system there will be several design patterns applied that may overlap. In such case, a class participates in more patterns. Each pattern contributes to its properties and behavior and it is impossible to name a pattern participant after the name given in a pattern catalogue. Pattern related naming is also excluded if one pattern is to be applied more than once within the same software system.

The above described requirements for pattern traceability and visibility cannot be in general met if a pattern is applied from currently used pattern presentation. Our motivation to represent patterns that allows to record pattern presence is even more compelling when we consider the current state in the area of pattern detection. It indicates that the presence of a pattern in the software system without any additional information is limited if possible at all [1, 8,11].

Our efforts to systemize pattern application come out also of the need to keep to constraints and regularities that participation in the pattern imposes on each participant. In all future development stages the pattern regularities, constraints and dependencies have to be hold so that the pattern essence is not broken.

1.2 Terminology

The term software pattern is being used excessively in almost every software area. One can think there is no need for another definition of patterns. But from the pattern application point of view, it seems to be essential to distinguish clearly among different meanings of the word pattern. A pattern refers to both the thing that lives in the code of a well-written program, and the instructions that help us understand or build that thing [4]. We feel it is reasonable to differentiate between them. The term *pattern instance* as addition to the term pattern is suggested. Whilst pattern according to our concept refers to a detailed description of a given solution to the problem and is typically published in a catalogue, pattern instance refers to those parts of an actually developed software that result from pattern application. A similar differentiation may be found in [5, 7, 9].

In addition to pattern and pattern instance, we introduce the term *pattern prototype*. Pattern prototype refers to the solution part of a pattern and is represented in the repository in a way suitable for pattern instantiation. Relationships among pattern, its prototype and instance are illustrated in Figure 1. Process of formulating pattern solution

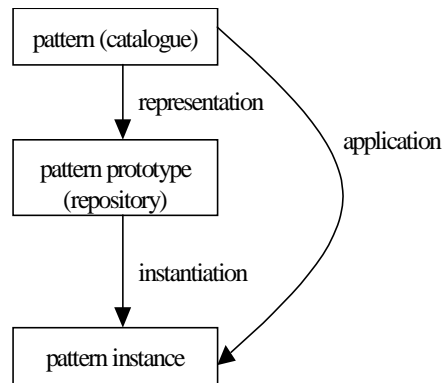


Fig. 1. Pattern application, representation and instantiation

essence is called pattern representation. From the pattern repository, a prototype may be chosen and instantiated into actual design so that pattern instance is created. In Figure 1, a difference between pattern instantiation and pattern application is illustrated. Application of design patterns is a process that starts from the pattern description in the catalogue and ends with formation of a pattern instance. Pattern application produces instances that are, as mentioned earlier, neither visible nor traceable. Our basic idea is to replace the current way of pattern application by a systematic approach when pattern prototypes are created first and from them pattern instances may arise in a controlled way. Visibility and traceability of patterns are key requirements for any tool support and would also improve readability and maintainability of software systems.

1.3 Bridge Pattern

As an example pattern that will be used throughout the paper, we show the Bridge design pattern. Main goal of the Bridge pattern is to decouple an abstraction from its implementation so that the two can vary independently [6]. Two separate class hierarchies, one for implementation and one for abstraction are bound through the "implementation" relationship that bridges them and lets them vary independently. The structure of the Bridge pattern is shown in Figure 2.

One particular problem that occurs when expressing pattern structure is that it does not show the multiplicity of some of pattern's participants. In the Bridge pattern, there can be for example several ConcreteImplementors but there are exactly two shown in pattern structure in Figure 2. Because of lack of expressiveness, one structure instead of many possible ones is given. This lack is complemented in other parts of design pattern where the multiplicity of some of pattern's participants is explained in textual form or code samples.

When a pattern is applied, those pattern parts with multiple occurrence are important. They are proposed to be called pattern parameters in [12]. In this approach, pattern parameters are flexible parts of the pattern structure that will be set to certain values when a pattern is

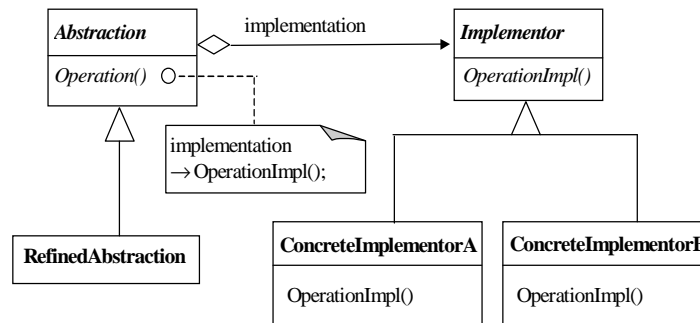


Fig. 2. The Bridge pattern structure

applied. Considering differences among pattern, pattern prototype and pattern instance, the pattern parameters as well as all pattern participants become free variables at the pattern prototype level, which in the process of pattern instantiation will be bound to specific values.

2 Pattern Prototype

Prototypes play a central role in our approach. Prototypes are not patterns, they focus only on the solution part of the pattern that is described mainly in its structure, participants, and collaboration. Because our concern is with pattern application, the solution part constitutes the core of our subject. Other, more informal parts of the pattern such as intent, motivation, or consequences, are not suitable for our representation. We are aware they provide important details for example in the process of identifying a suitable pattern to be incorporated into his/her design and suggest HTML files to complement the pattern prototype.

Our objective is to propose a pattern prototype representation that expresses pattern solution as a set of possible pattern structures with associated constraints.

2.1 Prototype Structure

Pattern prototype is a schema depicting all pattern relevant elements and their relations. Elements are depicted by a rectangle in the prototype and represent all those parts that constitute pattern solution. In the case of OO patterns, elements are classes and methods. It is in principle possible to extend the prototype so that less abstract details are expressed too, for example method parameters or method return values. Because of space limitations, less abstract elements are omitted here.

Relations between elements are depicted by a rhombus. A relation has its name, maximal and minimal cardinality, and an attribute that may specify what kind of relation it is dealt with. Those relations that have cardinality one to many (1:N) are pattern parameters. They cannot be specified to arbitrary values because setting a value to one parameter may impose constraints on other ones.

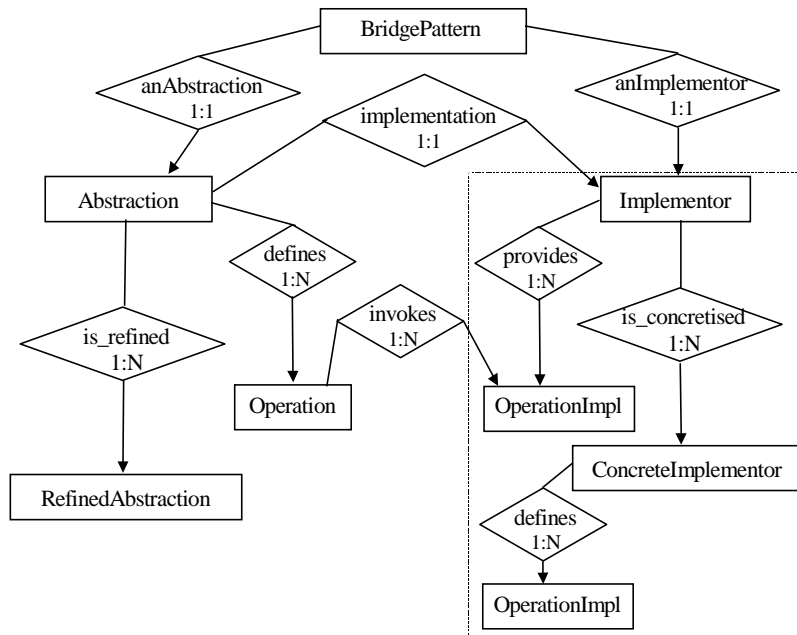


Fig. 3. Prototype for the Bridge pattern

Elements are in fact those roles that are to be played in a pattern. Relations connect the roles and determine dependencies among them. Each pattern prototype may be accompanied by pattern constraints. Constraints quantify pattern parameters and restrict possible pattern instances.

2.2 Example: Bridge Prototype

In Figure 3, a prototype for the Bridge pattern is shown. Elements and their relations constitute basic parts of the prototype. Abstraction, RefinedAbstraction, Implementor, and ConcreteImplementor are the prototype elements that correspond to main classes in the pattern. Abstraction and RefinedAbstraction are related through *is_refined* relation which has cardinality one to many because many RefinedAbstractions may be specified. Operation, OperationImpl, and RefinedOperation are method elements of the Bridge prototype. They are related to the class elements by *provides* and *defines* relations which are relations occurring in many design patterns. One for the Bridge pattern specific relation is *implementation*.

Besides the above elements and relations, the Bridge prototype contains also a pattern element named BridgePattern. This element is of particular importance because it not only makes pattern instance visible but it also allows many different and distinguishable pattern instances to be derived within the same software system.

A constraint that is attached to the Bridge prototype is as follows:

for each ConcreteImplementor

cardinality_of(Implementor.provides) = cardinality_of(ConcreteImplementor.defines)

It states that the number of OperationImpl for the Implementor should correspond to the number of OperationImpl for each ConcreteImplementor.

3 Pattern Instance

Pattern instances are design patterns applied in a particular design problem. Our technique proposes to build pattern prototypes first and instantiate them in a controlled way. Instantiation of pattern prototype produces a pattern instance that not only indicates pattern presence but also makes significant parts of the constituting pattern visible.

The process of pattern prototype instantiation implies that general prototype structure will be set to specific one by assigning the values to pattern parameters as well as by embedding the pattern prototype in the application domain. By instantiation, flexible parts of the pattern prototype are fixed and general names are replaced by application specific ones.

An important aspect is that the proposed instantiation supports multiple instances of the same prototype to be derived within the same software application. For each use of pattern prototype, a specific pattern instance is created that distinguishes them one from another. Besides this, constraints attached to the prototype enable to check structural dependencies in the pattern instance so that pattern regularities are obeyed.

3.1 Example: Bridge Pattern Instance

An instance of the Bridge pattern that occurs in the Lexi design is depicted in Figure 4. Lexi is a document editor introduced as a case study in [6].

The values to pattern parameters are assigned in the prototype instance, for example there are three ConcreteImplementors. All Lexi specific elements are typed after the roles they play in the pattern. This enables to determine the class/method role in the pattern. In addition, methods are becoming explicit. There also exists a special class for the pattern named LexiBridgePattern that records the pattern presence. This class points to the most important classes in the pattern. Relations are depicted uniformly in the diagram. The kind of relation is determined by its name and its specific multiplicity is explicitly stated.

Because constraints are attached to the prototype from which the instance has been created, structural dependencies may be examined.

4. Related and Future Work

One reason for proposing a pattern representation is to devise a tool that could support tasks of detection or application of design patterns. Tools that assist in pattern oriented software development have been discussed in several papers.

A precise visual specification for design patterns is proposed in [9]. Three different visual

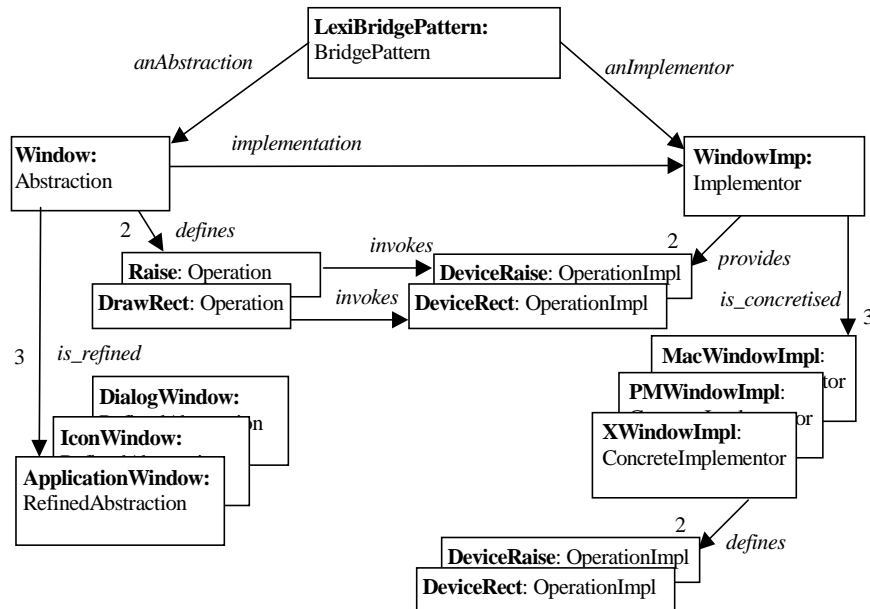


Fig. 4. The Bridge pattern instance in the Lexi document editor

models that represent a pattern at distinct level of detail permit the designer to operate with patterns at a higher level of abstraction without imprecision and ambiguity.

Unified Modeling Language [2] proposes collaboration diagrams for design pattern representation. In UML, design patterns are depicted as a dotted ellipse with lines connected to pattern related classes and objects. This approach does not support multiple occurrences of some pattern participants and dependencies among them. Also, to create several possible instances within the same software system may be troublesome.

An approach that relies on formulating patterns in a tangible and composable form is presented in [7]. An extension to Unified Modeling Language is proposed that depicts design pattern components in the notation similar to package notation. Different constituents that describe the design component can be displayed by zooming into detailed view.

In [5], every pattern is represented by a prototype consisting of a set of fragments that fulfil a particular role for the pattern. Unlike our approach they make no difference among pattern relevant fragment and all of them i.e., not only classes and methods, but also association relations, containment relations, and inheritance relations, are directly linked to the root pattern fragment.

In [10], a metalevel approach is chosen to represent patterns. A pattern is modelled as a metaschema that assembles components and relationships making up a pattern. A schema that is an instantiated metaschema, describe application dependent pattern instances. Through the FACE environment, the schema is given run-time semantics. Unlike our

schema, this approach makes explicit difference between class and method components.

In subsequent work, a catalogue of design patterns should be reformulated into a library of design pattern prototypes. A tool manipulating pattern prototypes is to be proposed in details and the way how pattern prototypes may be incorporated into a wider framework supporting software development based on design components will be investigated.

Acknowledgements

The work reported here was supported by Slovak Science Grant Agency, grant No. G1/4289/97.

References

1. Bansiya, J.: Automating Design-Pattern Identification. *Dr. Dobb's Journal* 6 (1998) 20-28
2. Booch, G., Jacobson, I., Rumbaugh, J.: *The Unified Modeling Language User Guide*. Addison-Wesley (1998)
3. Budinsky, F. J., Finnie, M. A., Vlissides, J. M., Yu, P. S.: Automatic Code Generation From Design Patterns. *IBM Systems Journal* 2 (1996)
4. Coplien, J.: Progress on Patterns: Highlights of PLoP'94. In: *Object Expo Europe*, London (1994)
5. Florijn, G., Meijers, M., vanWinsen, P.: Tool Support for Object-Oriented Patterns. In: Aksit, M., Matsuoka, S. (eds.): *ECOOP'97*, Jyväskylä, *Lecture Notes in Computer Science*, Vol. 1241. Springer-Verlag (1997) 472-495
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman (1995)
7. Keller, R. D., Schauer, R.: Design Components: Towards Software Composition at the Design Level. In: *Proc. of the 20th International Conference on Software Engineering*, Kyoto, IEEE Computer Society (1998) 302-311
8. Krämer, Ch., Prechelt, L.: Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In: *Proc. Working Conference on Reverse Engineering*, Monterey, IEEE Computer Society Press (1996) 208-215
9. Lauder, A., Kent, S.: Precise Visual Specification of Design Patterns. In: Jul, E. (ed.): *ECOOP'98 – Object-Oriented Programming*, *Lecture Notes in Computer Science*, Vol. 1445. Springer-Verlag (1998) 114-134
10. Meijler, T. D., Demeyer, S., Engel, R.: Making Design Patterns Explicit in FACE, a Framework Adaptive Composition Environment. In: Jazayeri, M., Schauer, H. (eds.): *ECES/FSE'97*, *Lecture Notes in Computer Science*, Vol. 1301. Springer-Verlag (1997) 94-110
11. Seemann, J., Wolff von Gudenberg, J.: Pattern-Based Design Recovery of Java Software. In: *Proc. of 6th International Symposium on the Foundation of Software Engineering*, ACM SIGSOFT 6 (1998) 10-16
12. Smolárová, M., Návrát, P., Bieliková, M.: Abstracting and Generalising with Design Patterns. In: U. Gündükbay, U., Dayar, T., Gürsay, A., Gelenbe, E. (eds.): *ISCIS'98*, 13th Int. Symposium on Computer and Information Sciences, Belek-Antalya, Turkey, IOS Press Ohmsha (1998) 551-558