Chapter 3
Design Patterns: Applications and Open Issues

K. Lano

Abstract The field of software design patterns has grown extensively since the first work on patterns in the 1990s. Design patterns have proved useful as encodings of good design practice and expert knowledge in a wide variety of domains, from enterprise information systems to software security. We look at some recent developments in the application of patterns, and identify some remaining theoretical and practical issues with the use of patterns.

1 Introduction

The concept of software design patterns originated in the late 1980s and early 1990s, based on analogy with the architectural design patterns which had been formulated by Christopher Alexander in his series of books [1, 2].

The key text introducing software design patterns to software engineers and developers was "Design Patterns" by Gamma, Helm, Vlissides, Johnson, published in 1994 [7]. This introduced 23 patterns, such as Observer, Visitor, Singleton, Iterator, Template Method, classified into the three general categories of Creational, Structural and Behavioural patterns. Subsequently there has been widespread identification and application of patterns in a wide range of software domains.

Classic design patterns include:

– Template method, Observer, Strategy (Behavioural)
– Facade, Adapter (Structural)
– Singleton, Builder (Creational).

A design pattern expresses a characteristic solution to a common design problem: the pattern describes classes, objects, methods and behaviour which constitute the

K. Lano (✉)
Department of Informatics, King's College London, London, UK
e-mail: kevin.lano@kcl.ac.uk

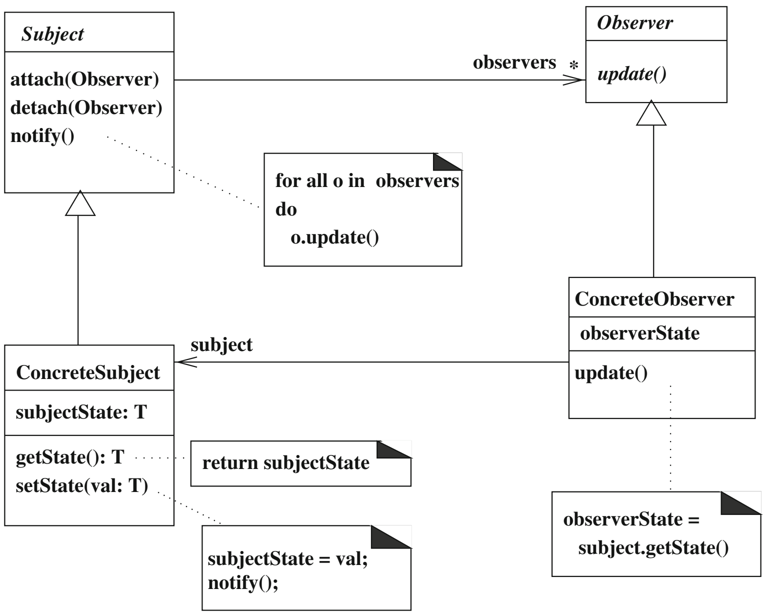

Fig. 1 Observer pattern

solution. The solution is intended to be an improvement on the system without the pattern: more flexible and adaptable, more modular, easier to understand, etc.

For example, the Observer pattern describes a way to organise a system which involves a data source (of arbitrary complexity) and multiple views or representations of that data. The pattern separates out the data and views into independent hierarchies of classes, connected by an association which represents the observation relationship (Fig. 1).

The pattern improves the modularity of the system by separating the system code into two distinct components which have clearly distinguished responsibilities: the data component (Subject) is purely concerned with internal data management, whilst the view component (Observer) is purely concerned with presentation of the data. The pattern improves the adaptability and extensibility of the system by using inheritance: new forms of data and new kinds of view can be added to the system without changing the observer mechanism, provided that the data and view classes conform to the Subject and Observer interfaces. On the other hand, the pattern has potentially negative implications for efficiency, because all interaction between the data and views has to take place via method calls. This is a common problem with many design patterns: improvements in the logical structure of a system may reduce efficiency. However it is generally considered that the gains are more significant for software quality—and therefore long-term reductions in software maintenance costs—than the loss of optimal performance.

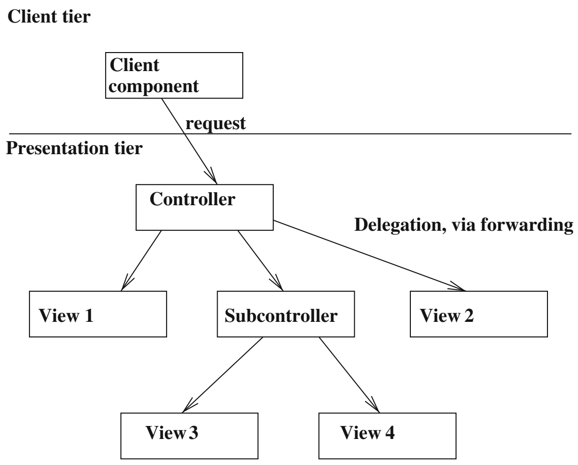

Fig. 2 Front controller pattern

Patterns are distinguished from *idioms*: small-scale repetitive structures of code, such as the standard for-loop header in C. They are also distinguished from *refactorings* of programs or models: small-scale and incremental structural transformations of systems, eg.: pulling up attributes from a subclass to a superclass. However, in some cases refactorings can be used as steps by which a pattern can be introduced [10].

2 Specialised Design Patterns

Following the identification and formulation of a large collection of general purpose software patterns, work began on the identification of patterns specialised for particular domains. For example, in [13], a collection of patterns aimed at improving the design of enterprise information systems (EIS) is described. The patterns serve an educational purpose, transferring some expertise in this complex domain from experienced developers to those unfamiliar with its particular problems and solutions. EIS patterns include Front Controller (Fig. 2), Intercepting Filter, Value Object, Data Access Object, etc.

These patterns are in some cases specialisations of classical patterns (e.g.: Intercepting Filter can be regarded as a special case of Chain of Responsibility, and Front Controller as a version of Facade), or they may be specific to the domain (e.g.: Value Object).

Likewise, in the domain of service-oriented architectures (SOA), patterns for services, such as Broker, Router, etc. have been formulated, and for security concerns there are patterns such as Access Proxy [9]. Patterns have also been recognised at the specification and analysis stages, e.g.: the Scenario pattern of [5]. More recently, patterns have been defined for model transformations, such as Auxiliary Metamodel, Phased Construction, etc. [15].

3 Design Patterns in Model-Driven Development

Model-driven development (MDD) emphasises the use of models such as UML class diagrams and state machines as the key documents of a software system development, and aims to raise the level of abstraction in system development away from platform-specific and programming language-specific coding towards business-level specifications and platform-independent designs. Design patterns certainly have an important role in model-driven development, as a platform-independent technique which can be applied to specifications and analysis models as well as to language-independent designs.

A key element of model-driven development are *model transformations*, which are used to map models from one level of abstraction to another (e.g.: generation of a design from a specification, or of a program in a particular language from a design), or to restructure, filter or combine models.

There is a two-way relationship between design patterns and model transformations: a design pattern can be seen as a kind of model or program transformation, defining how a system can be rewritten from an initial structure that is deficient or of poor quality in some sense, to an improved version. Such a transformation is termed a restructuring or refactoring transformation. On the other hand, transformations themselves can be the subject of patterns.

In the first case, patterns have been used in semi-automated MDD to guide development by identifying good design choices [11]. Patterns have also been incorporated into programming languages and frameworks, such as the use of Iterator within Java, C# and C++ libraries, and of Observer/Model-View-Controller in internet application frameworks.

Model transformation patterns, together with quality measures of transformations, have been used to guide model transformation development, eg., to define modular designs, and to optimise transformations [15].

An example of a model transformation design pattern is the Phased Construction pattern of [15], shown in Fig. 3.

This pattern defines a particular style of organisation for a transformation specification, where successive layers of the structure of the target model are constructed by successive rules of the transformation. This contrasts with a recursive style of specification in which a rule may carry out the mapping of all layers of structure navigable from a particular object. The phased construction pattern improves the modularity of the specification.

4 Design Pattern Formalisation and Verification

Design patterns are usually described using text, diagrams such as UML class or interaction diagrams, and code. A standard form of pattern descriptions has evolved from the Gamma et al. book onwards, and usually involves a description of the

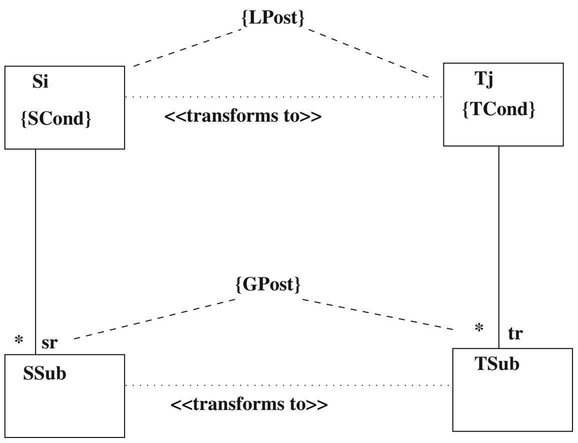

Fig. 3 Phased construction pattern

problem that the pattern is designed to solve, positive and negative indicators for using the pattern, and a description of how to introduce the pattern, together with a rationale for using it.

While such informal descriptions are useful to communicate the concepts and purpose of the pattern, they are insufficient to support automated selection and application of the pattern, or to support verification that the introduction of the pattern preserves desirable properties of the system, or that it actually improves some measure of quality such as modularity.

Approaches for pattern formalisation include [3, 4], which characterises pattern instances using a first-order predicate over the UML metamodel. For example, to specify that there is an instance of Template Method in a system, the predicate asserts that there is an abstract class in the set *classes* of classes of the system, that it has a finalised (leaf) template method which calls non-leaf operations, etc. This approach enables checking of a model for conformance to a pattern, but does not represent the process of introducing a pattern as a design improvement. A related approach using metamodelling to represent patterns and their applications is [11]. In [12] we characterise design patterns as model transformations, based on a modal logic semantics for UML.

Patterns are difficult to formalise because they often have considerable variability: the Template Method pattern for example could apply to any number of classes in the initial system model which have the same operation, and the factoring out of common code from these classes could be carried out in many alternative ways.

However, despite these limitations, we consider that it is of benefit to try to formalise patterns as far as possible, and to do so in a manner which supports their automated selection and application, and which supports their verification. We suggest that:

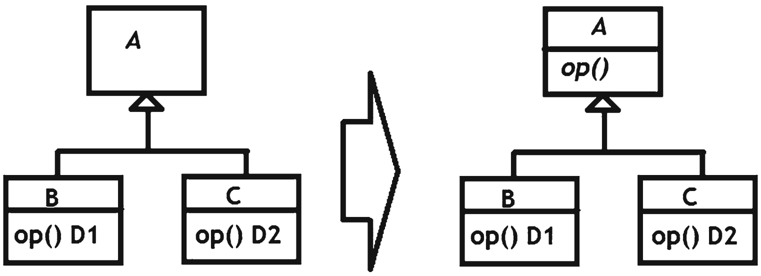

Fig. 4 Pull-up method refactoring

1. Measures of system quality should be defined by which pattern applications can be selected: values of measures which indicate a poor quality aspect in a system will identify that particular patterns should be applied to improve this measure; if there are several possible alternative pattern applications, then one which maximally improves the measure should be chosen.
2. Pattern applications should be formalised as model transformations, usually as update-in-place restructuring transformations.

This approach models both the *purpose* of a design pattern (improvement in particular quality measures) and the *process* of introducing a pattern, as a model transformation.

A range of quality measures could be defined, which characterise the modularity, complexity, degree of code duplication, data dependency and operation dependency structure of the system.

For example, two measures of a design-level UML class diagram which could be formulated are:

1. The number of classes A which have two or more subclasses, and where all these subclasses have a common operation $op(T1) : T2$ (possibly with different definitions, but with the same signature), but A does not have this operation defined.
2. The sum of all syntactic complexities of method definitions, e.g.: the sum of the number of features, operators and statements in their code, if expressed in a pseudocode programming language.

Non-zero values for measure 1 will identify cases where commonalities of subclasses have been incompletely recognised (e.g.: the LHS of Fig. 4), and indicate the application of the 'Pull up method' refactoring [6], to produce a system with an improved measure (RHS of Fig. 4).

If the definitions $D1$ and $D2$ have no common subcode, then measure 2 cannot be improved. However, if they share some small segment $D0$ of code, then a single definition of this code can be factored out and placed in A, reducing measure 2 by $complexity(D0)$, in the case of two subclasses, and by $(n - 1) * complexity(D0)$ if there are n subclasses with the duplicated code. Finally, if the majority of the code of $D1$ and $D2$ is in common, then the factorisation can be inverted, which leads

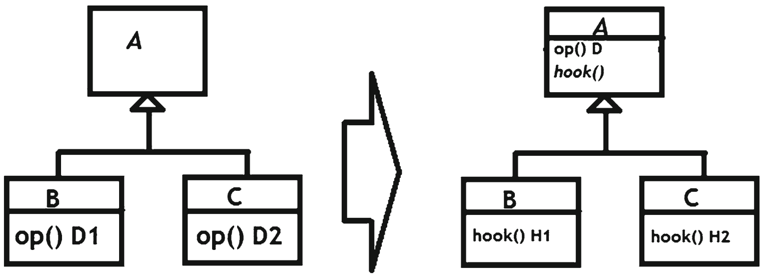

Fig. 5 Template method pattern application

to application of the Template Method pattern (Fig. 5), and a reduction in overall complexity of $complexity(D)$ or $(n - 1) * complexity(D)$.

The introduction of a pattern can be considered as the application of a transformation rule, of the general form:

$$\begin{aligned}&\textit{if pattern applicable to model part} \quad \Longrightarrow \\ &\qquad\qquad \textit{rewrite model part according to pattern}\end{aligned}$$

In the case of Template Method, the pattern is recognised as applicable if there is a situation shown on the LHS of Fig. 5 where the common parts of $D1$ and $D2$ have a larger complexity than the distinct parts. The application of the pattern should factor out the maximum common part D of $D1$ and $D2$ in order to maximise the reduction in measure 2. Likewise, Front Controller should be introduced if there is duplicated request checking code in two or more presentation tier request target components. The duplicated code should be factored out of the maximal number of such components possible.

In [15] we give a set of measures and design patterns for transformation specifications, and give rules for selection of patterns based on the measures. For example, an alternation of quantifiers value greater than 0 in a constraint conclusion suggests applying the "Phased construction" pattern to the constraint, in order to remove this alternation, which complicates analysis, adaption and verification of the constraint. Similarly, if an expression of more than a given critical complexity occurs in two or more separate places in a specification, it should be factored out as a called operation: the larger the expression complexity and the higher the number of occurrences, the higher is the priority for application of this refactoring.

4.1 Pattern Verification

If we consider pattern introduction as a form of model transformation, then definitions of transformation correctness can be adopted for design patterns. The following correctness definitions are adapted from [14]:

Semantic preservation of φ: if the original system m satisfies a property φ so does the transformed model n:

$$m \models \varphi \implies n \models \varphi$$

For example, factoring out code of an operation by applying Template Method does not change the pre-post semantics of the operation.

Syntactic correctness: the transformed model satisfies necessary restrictions of the modelling/programming language. For example, that Template Method cannot introduce name conflicts of features into classes: in order to ensure this, names of hook methods should be chosen appropriately to avoid any existing method names.

Termination: that an automated series of pattern applications will eventually terminate.

In fact the use of measures will help to ensure termination: if each pattern application definitely reduces one measure and does not increase any other, then eventually the process will terminate with no pattern being applicable (assuming that measure values are always non-negative integers).

Confluence is also important in some cases, i.e.: a guarantee of the uniqueness of models resulting from a transformation process is required. One technique to ensure this is to show that the quality measures attain their minimal (best) values in an essentually unique situation, amongst those which can be reached by applying the rules. This holds for Pull up Methods and measure 1, for example.

5 Conclusions

Patterns have been a very productive area of research in software engineering, with many practical applications and benefits. The use of patterns is consistent with Model-driven development in raising the level of abstraction in software development from low-level coding to the level of concepts and ideas.

It is still an open-ended issue of how best to describe and formalise design patterns, and how to automate/semi-automate the selection and application of patterns. In addition, the general verification of the correctness of pattern applications is not solved.

In this chapter we have surveyed the research into these issues, and we have proposed that design patterns should be treated as model transformations for the purpose of formalisation and verification, and that measures should be defined to support the detection and selection of patterns, and to provide evidence of improvements obtained by introducing a pattern.

References

1. Alexander C. A Pattern language: towns, buildings, construction. New York: Oxford University Press; 1977.
2. Alexander C. The timeless way of building, New York: Oxford University Press; 1979.
3. Bayley I, Zhu H. Formalising design patterns in predicate logic, SEFM '07. Taiwan: IEEE Press; 2007.
4. Bayley I, Zhu H. Specifying behavioural features of design patterns in first order logic COMPSAC '08. Washington: IEEE Press; 2008.
5. Fowler M. Analysis patterns: reusable object models. Boston: Addison-Wesley; 1997.
6. Fowler M. Refactoring: improving the design of existing code. Boston: Addison-Wesley; 2000.
7. Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley; 1994.
8. Grand M. Patterns in Java. New York: John Wiley & Sons, Inc; 1998.
9. Hafiz M. Security pattern catalog. http://www.munawarhafiz.com/securitypatterncatalog/index.php 2013.
10. Kerievsky J. Refactoring to patterns. Reading: Addison Wesley; 2004.
11. Kim D. Software quality improvement via pattern-based model refactoring, 11th IEEE high assurance systems engineering symposium. Washington: IEEE Press; 2008.
12. Lano K. Formalising design patterns as model transformations. In: Taibi T. editor. Design pattern formalisation techniques. Hershey, PA: IGI Press; 2007.
13. Lano K. Model-driven software development with UML and Java. London: Cengage Learning; 2009.
14. Lano K, Kolahdouz-Rahimi S, Clark T. Comparing verification techniques for model transformations. MODELS: Modevva workshop; 2012.
15. Lano K. Kolahdouz-Rahimi S. Optimising model-transformations using design patterns. MODELSWARD: 2013.
16. Massoni T, Gheyi R., Borba P. Formal refactoring for UML class diagrams, 19th Brazilian symposium on software engineering. Uberlandia: 2005.