Can Styles Improve Architectural Pattern Reuse?

Sorana Cîmpan  
LISTIC - Université de Savoie, France  
Sorana.Cimpan@univ-savoie.fr

Vincent Couturier  
LISTIC - Université de Savoie, France  
Vincent.Couturier@univ-savoie.fr

Abstract

Software patterns are generic solutions to recurring problems. They are represented as a triplet (problem, context, solution). Several kinds of software patterns were identified, examples of which are analysis, architecture, design or implementation patterns. The current practice is to represent patterns as documents that communicate best practices. Their reuse is limited by several factors among which the lack of formalization. In the context of architecture centered software development, architectural styles are means to represent families of software systems. Several architecture description languages offer support for representing architectural styles. We believe that the use of architectural styles in the representation of architectural pattern solutions can improve the pattern reuse level. The paper details how this improvement can be achieved.

1. Introduction

Starting from Christopher Alexander’s work[3][4], patterns have been used and adapted to several application domains, among which software engineering and information systems engineering. They allow the description of reusable solution elements, as well as knowledge and know-how capitalization. In Alexander’s proposition, a pattern describes a problem that frequently appears in an environment as well as a solution that can be used differently several times. Others see patterns as abstractions used by design or code experts that provide solutions in different phases of software engineering[11].

Patterns can be divided in four categories: analysis patterns, architectural patterns, design patterns, and idioms, the latter also being known as implementation patterns[21].

Analysis or conceptual patterns, according to Riehle and Züllinghoven [23] are used to describe solutions related to problems that arise during both the requirement analysis and the conceptual data modeling phases. Analysis patterns reflect rather conceptual structures of the application domain and are not concerned with computational solutions.

Architectural and design patterns are both related to the design process. Though, they differ in the level of abstraction where each one is applied. Architectural patterns express a fundamental structural organization schema for software systems and can be considered as templates for concrete software architectures [7]. Design patterns [12] provide schemes to refine the subsystems or components of a software system and thus are more abstract (and of smaller granularity) than architectural patterns.

Idioms [10] are used in the level of programming languages and deal with the implementation of particular design issues.

In this paper we focus on architectural patterns. As it will be later detailed, the pattern reuse is limited by several factors. We believe the use of research results related to software architectures can alleviate this problem.

Software architecture research aims at reducing development costs by identifying commonalities among closely related products. Software system architecture entails the principal design decisions concerning the system, and is rather orthogonal to the development process. Architectural models group some or all of the architectural design decisions. The architecture is thus a mean for capitalizing design knowledge, as the same architecture can be used for different systems or serve as a base for the construction of a new architecture. A step further in the design knowledge capitalization (and thus reuse) can be made by the use of architectural styles. The latter are means of capturing families of architectures [18], and can be seen as a set of design rules that guide and constrain the development of software architectures.

Several architecture description languages allow the description of architectural styles. ACME [13] allows the definition of architecture families. Armani [19] is an extension of ACME for style definition, allowing the specification of several kinds of constraints. While these two languages take only the structure of the system into account, Wright [1] (and its
extension Dynamic Wright [5]), as well as the ArchWare ADL[20] and Architectural Style Language [9] take into account the system behavior. They are both based on process algebras: Wright on CSP, ArchWare on π-calculus.

The way styles are defined is obviously constrained by the mechanisms provided in the language used to define them.

We believe that the use of architectural styles in the representation of architectural patterns solutions can improve their reuse.

The paper is organized as follows. Section 2 presents the different pattern modeling approaches. The process of using such patterns (pattern reuse) is presented in section 3. Section 4 presents the kind of formalization needed to improve architectural patterns reuse. The paper concludes with discussions.

2. Pattern modeling

A pattern is dedicated to knowledge and know-how appropriation by human actors. Thus, it is not a compilable code but a description expressed by a text (i.e., natural language), a diagramming language or a formal language.

2.1. Textual documentation

Natural languages “share the great advantage for modeling of being the most expressive communication method available, and the great disadvantage of being the most expressive communication method available”, and are therefore inherently ambiguous [1]. Another advantage of natural languages is that they involve no overhead to learn (for adults). Other disadvantages are that they are often more verbose than other modeling alternatives (“a picture is worth a thousand words”), and that they are difficult to impose structure or stick to conventions.

Many patterns are only based on natural language for their documentation and are thus reusable with more difficulties. Others complete their textual representation with one of the two following techniques.

2.2. Documenting patterns with semi-formal notations

We limit our analysis to semi-formal representations of pattern solution based on diagrams. Diagramming languages structure information, facilitate learning regardless of the representation and reduce ambiguities by using a more precise notation when applying patterns. Moreover, the use of a unique modeling language by pattern writers and pattern-based software engineers allows sharing of a common understanding, facilitates appropriation and improves pattern reuse. This kind of formalization has contributed to the success of patterns.

When documenting the structure of a pattern, developers usually use the object oriented paradigm and UML class diagrams. As standard UML class diagrams don't always allow to represent the entire information needed, some approaches use UML extension mechanisms, such as stereotypes, constraints, or tagged values to improve the representation of patterns and their reuse[24].

2.3. Formal description of patterns

According to Blazy and al. [6], much of the behavior of solution elements inside patterns can be modeled using formal languages of one sort or another (e.g., B, Z or VDM). These languages have the advantage of precise and verifiable semantics. However, mathematical languages such as B, Z or VDM are not intuitive. As most pattern writers or pattern-based software engineers are not familiar with them, their adoption demands an amount of effort which limits their reuse.

3. Pattern reuse

Pattern reuse, also called pattern-based software engineering, integrates three activities:
- selection of relevant patterns (for instance, by using a patterns map),
- adaptation to user context or copy of each pattern,
- and possibly code generation.

The fist activity consists in analyzing a problem in a given context and selecting relevant patterns that match them. Maps [8] give an overview of the set of patterns potentially applicable and can be used during this activity in order to help software engineers to sort and select them.

The second activity consists in copying or adapting pattern solution elements to user context (instantiation). Copy is used when solution elements are expressed in a textual form and consists in reusing solution without any modification. Adapting solution elements to the design in progress is mainly used when these elements are written according to the object-oriented approach (e.g., UML notation). According to Prieto-Diaz [1], three basic adaptation mechanisms exist for reusing patterns in order to develop an application:

- Instantiation, also called imitation, allows
different elements of a pattern solution to be renamed.

- **Composition**, also called integration, associates two or more patterns. Different kinds of association can be defined.
- **Extension**, allows new elements to be added to existing patterns or, existing elements to be removed or modified. These elements can be attributes, methods or associations.

The third activity aims at generating code from pattern instances. A subset of design patterns and idioms can only be concerned by this phase. Several approaches to pattern-based code generation were proposed [14][1].

4. **What kind of formalization to improve architectural patterns reuse?**

The degree of pattern formalization influences the way the three pattern reuse activities are conducted. The more formal the language used, the less ambiguous the description of the pattern, which can ease the selection activity. Moreover, semi-formal and formal notations often propose mechanisms that ease the second and the third activity. Nevertheless, formal methods are not necessarily the straight answer to our problem. The main drawback relies on their sometimes counter-intuitive nature, which limits their adoption. Moreover, as “a picture is worth a thousand words”, semi-formal methods lead to pattern definitions which are easier to get onto. Thus we believe that in order to improve pattern reuse, both semi-formal and formal modeling techniques have to be considered. A textual part is always provided to describe the problem and the context. A pattern solution entails thus two parts:

- The semi-formal description allows the pattern user to get onto the pattern and understand the solution overall structure
- The formal description allows to have a more detailed definition (and thus less ambiguous), and benefits from mechanisms that ease the instantiation

The problem of the choice between several modeling techniques is not particular to patterns, and is encountered in every design activity. The same issue has been addressed by the software architecture community, which among others, worked on finding the appropriate modeling techniques for describing software architectures. Several architectural description languages have been proposed by the research community [17] with a broad variety in their formal nature, while the practicing community used boxes and lines (or UML) to represent software architecture.

Faced to the weak acceptance of ADLs by the practicing community, researchers questioned the use of specific languages as opposed to more generic approaches, such as UML [15][16]. Some ADLs proposed UML-based concrete notations to complement the textual ones [2].

Although, in practice diagramming languages are often used to describe software architectures, they lack the semantics needed to represent complex interactions between object collections. Even if UML 2.0 introduces the notion of software component (COM, EJB component, …), it is in general very hard to structure architectural elements by using the three UML concepts: component, connector and architecture. Medvidovic and Hofmeister [15][16] have identified several drawbacks when using UML models or extension mechanisms to represent software architectures and their components.

5. **Discussion**

Architectural styles and architectural patterns can be considered as two complementary modeling techniques allowing the encapsulation and reuse of architectural design expertise. On the one hand, styles provide a set of constraints that govern the interaction of system components. On the other hand, patterns help designers in better structuring an environment representation, guide them in the perception of the real world and provide a high abstraction level. Patterns allow getting onto complex systems in a simple manner, without needing a thorough study of their implementation.

Formal and semi-formal methods offer complementary means for the specification of software architectures: rigor and verification for the first, readability and validation for the second. We believe integrating styles as means to formalize architectural pattern, complementary to a UML 2.0 description increases the potential reuse of such patterns.

The first obvious drawback of adopting this approach is the need for the pattern user to master yet another language. This drawback can be balanced by a wise choice of the style definition language, so that the gains are worth the effort. Moreover, despite their sometimes formal base, such languages use intuitive constructs.

We believe the following capabilities should be analyzed when selecting a style definition language for pattern formalization:

- ability to represent structural constraints on the architectures;
- ability to represent behavioral constraints on the architectures;
• possibility to use template or constructor mechanisms such that architectures following the styles can be generated from the style; such mechanisms ease the pattern adaptation to a specific context.

Using the style definition of a pattern, architectures that follow the pattern are constructed in a given ADL (related to the style definition language). The choice of this ADL is also important, as it also has impact on how reusable the pattern definition is.

The first gain is obviously the existence of the architecture definition itself, representing the application for whose construction the pattern is used. These architecture guides the pattern user in its application development. Nevertheless, several tools related to the selected ADL allow further more to ease the pattern reuse:

• architecture animators provide simulations of the system behavior: such animations are possible when the ADLs represent behavior, and a virtual machine is provided; the use of such a tool allows to validate the architecture with respect to its intended behavior;

• code source generator: when available, they generate the entire or part of the code source, which of course in terms of reuse is obviously a gain.

This paper’s title is a question: can ADLs (and more precisely styles) improve pattern reuse? We believe so, and we are currently experimenting the use of ArchWare ASL [9] in the description of several information system architectural patterns.

6. References


[12]. E. Gamma, R. Johnson, R. Helm, J. Vlissides, Design patterns, elements of reusable object-oriented software, Addison-Wesley, 1995.


