The Journal of Systems and Software 110 (2015) 136–154

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

A goal-oriented approach for representing and using design patterns

Luca Sabatucci[a,*], Massimo Cossentino[a], Angelo Susi[b]

[a] ICAR-CNR, Palermo, Italy
[b] Fondazione Bruno Kessler, Trento, Italy

ARTICLE INFO

Article history:
Received 13 February 2014
Revised 27 July 2015
Accepted 29 July 2015
Available online 11 August 2015

Keywords:
Design patterns
Goal modeling
Goal reasoning

ABSTRACT

Design patterns are known as proven solutions to recurring design problems. The role of pattern documentation format is to transfer experience thus making pattern employment a viable technique. This research line proposes a goal-oriented pattern documentation that highlights decision-relevant information. The contribution of this paper is twofold. First, it presents a semi-structural visual notation that visualizes context, forces, alternative solutions and consequences in a compact format. Second, it introduces a systematic reuse process, in which the use of goal-oriented patterns aids the practitioner in selecting and customizing design patterns. An empirical study has been conducted the results of which supports the hypothesis that the goal-oriented format provides benefits for the practitioner. The experiment revealed a trend in which solutions better address requirements when the subjects are equipped with the new pattern documentation.

1. Introduction

Software patterns are known as proven solutions to recurring problems in the design and the implementation of software systems Buschmann et al. (1996). This common definition has been refined many times over the years. An interesting definition mentions patterns as instruments for taking decisions during software development Zdun (2007); Gross and Yu (2001); Araujo and Weiss (2002).

The importance of this observation is that the quality of a software product is highly dependent on the design phase in which strategic decisions are made that remain with the system for the rest of the development. Bad design decisions generally negatively affect the final product Yacoub et al. (2000). Software patterns help inexpert developers to assess the impact of a decision when the final product is not mature enough to evaluate if a decision is good or not Yacoub et al. (2000).

Since their invention, the response of the research community has been enthusiastic: practitioners have assisted to a phenomenon of proliferation of the categories of patterns, and to an impressive number of collections of patterns addressing a fairly extensive set of problem domains Henninger and Corrêa (2007); Rising (2000); Cline (1996). Patterns exist for solving architectural issues Buschmann et al. (1996), object-oriented design Gamma et al. (1995), coordination and process problems van Der Aalst et al. (2003), parallel and concurrency execution Schmidt et al. (2013), security concerns Schumacher et al. (2013) and so on. For instance, the Pattern Almanac Rising (2000), published in the year 2000, contains over 1000 patterns. Such a proliferation also generated many duplicates, i.e. patterns that are variants of the same design principle Henninger and Corrêa (2007). For instance the Extended Observer UIUC (0000) and the Middle Observer Iaría and Chesini (1998) consider specific application contexts of the original Observer pattern Gamma et al. (1995).

The value of design patterns is that of being the result of experience on the field gained over several years of trial-and-error attempts. Using a pattern during software development consists in exploiting a well-proven solution, with general benefits on software quality Riehle (2011) and on maintenance process Prechelt et al. (2002); Duell (2001). Nevertheless the use of patterns in software development practices is far from being trivial. A software pattern is generally less tangible but more flexible than code. A class library provides a collection of classes and methods to use in a black-box fashion. Conversely a pattern is generally described by highlighting a relation between a certain context, a problem, and a solution Buschmann et al. (1996), and it specifies a level of abstraction that is above the level of classes or components. Therefore a pattern is harder to use because its abstractions must first be understood and later be instantiated in specific problem Riehle (2011).

The most common format for the documentation of patterns is basically text with some visual support and code examples. Advantages are in the richness and flexibility of the natural language and the way it fosters human creativity. Shortcomings are in the potential ambiguity and the average length (in number of pages) necessary

* Corresponding author. Tel.: +393936478191.
E-mail addresses: sabatucci@pa.icar.cnr.it (L. Sabatucci), cossentino@pa.icar.cnr.it (M. Cossentino), susi@fbk.eu (A. Susi).

http://dx.doi.org/10.1016/j.jss.2015.07.040
0164-1212/© 2015 Elsevier Inc. All rights reserved.

for documenting all the details that tend to be spread among many sections of the documentation.

The research community has spent much effort in improving pattern documentation by increasing the level of formalization and the integration with design processes and techniques Zdun (2007); Gross and Yu (2001); Mikkonen (1998); Yacoub and Ammar (2001). Nevertheless most of these approaches are not able to fully represent the abundance of details (for instance they fail to represent alternative and decision points in the solution). In addition, mainly focusing on the solution side, they raise the risk that practitioners confuse a design pattern with its structure diagram Riehle (2011).

The overall objective of this paper is to propose a novel approach for describing patterns that, on one hand, preserve all the details and, on the other hand, makes accessible the decision-relevant information such as motivation, alternatives, consequences, and forces.

The idea is that, regardless of the category of the pattern, the common element of many pattern descriptions is the *design rationale*, i.e. a set of design goals, design decisions and expected consequences in terms of software qualities. In other words, applying the patterns is similar to making a design decision, which is a cognitive process concerning forces to balance Zdun (2007); Gross and Yu (2001) and design decisions to take McPhail and Deugo (2001); Harrison et al. (2007) in order to configure the elements of the system for solving a specific design problem.

Our first contribution is a goal-oriented approach for documenting patterns based on the *i** strategic modeling Yu (1996), a conceptual framework for modeling cognitive processes and strategic contexts. The main concepts of *i** are exploited for representing, in a semi-formal notation, a software pattern by preserving all the details that the textual format provides. The *i** notation was built for modeling strategic dependencies in the context of requirements engineering, but it is general enough to be used in many other organizational application settings. By exploiting this general-purpose nature, the goal-oriented pattern documentation was constructed to be independent of the programming paradigm and the specific category of the pattern. Despite the fact that the notation revealed itself to be expressive enough to document a number of software patterns belonging to different categories existing in literature Iaría and Chesini (1998); Riehle (1998); Wallingford (1997), this paper primarily shows examples of architectural and design patterns taken from the GoF's book Gamma et al. (1995) or from the Pattern Oriented Software Architecture book series Buschmann et al. (1996). Only one example of a workflow pattern van Der Aalst et al. (2003) has been added for discussing independence from the domain.

The second contribution is a systematic process in which the goal-oriented documentation format plays a central role during the software development. The process includes guidelines for a methodical exploration of context problem and forces with the aim of improve the pattern selection activity. In addition the process provides guidelines for driving the practitioner to focus on the design decisions to take for customizing its solution for the specific problem context.

Finally, we report an empirical study conducted to investigate to which extent the goal-oriented format and the related process may offer benefits for practitioner. We empirically observed that class diagrams, created through pattern employment, better meet requirements when inexpert designers are equipped with the goal-oriented documentation format of the patterns. We argue that describing patterns through goals discourages the bad practice of using solutions as code templates; conversely it fosters developers to reason with high level concepts that the pattern embeds and it increases their ability to customize the pattern for the specific problem domain.

This paper is organized as follows. Section 2 analyzes the state of the art and sets up the background for the proposed approach. Section 3 presents the notation used for the goal-oriented pattern documentation. Section 4 illustrates the employment of goal-oriented patterns in a systematic process in which goals aids at discovering which one to use and how to customize it for the specific problem context. Section 5 illustrates an experiment to substantiate our claims. Some remarks about the proposed approach are reported in Section 6 and finally, conclusions are drawn in Section 7. Three complete examples of goal-oriented pattern descriptions are reported in Appendix.

2. Background for the proposed approach

The most common format for the documentation of patterns is basically natural language with some visual support and code examples. Depending on the category of pattern the documentation structure slightly changes including a different set of sections. For instance a GoF's pattern is documented through name, intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code, known uses and related patterns. On the other side, a POSA pattern does not explicitly include intent, collaborations and participants, adding a summary, a solution, dynamics and variants. The strengths of such format are due to the natural language: the verbosity and the flexibility allow conveying complex abstractions. This format has been conceived to foster interpretation and creativity. The shortcomings are the potential ambiguity of the natural language, the average length (in number of pages) and the redundancy of details that are spread among many sections of the documentation.

The design pattern community spent much effort in improving the pattern documentation by raising the level of formalization and the integration with design process and techniques. It is possible to mention declarative Eden et al. (1998); Mapelsden et al. (2002), formal Mikkonen (1998), UML based Yacoub and Ammar (2001) and semantics approaches Zdun (2007); Gross and Yu (2001); Araujo and Weiss (2002); Sabatucci et al. (2009). A review is proposed below.

2.1. Formal and semi-formal pattern documentation

Eden et al. propose LePUS Eden et al. (1998), a declarative pattern specification language that uses higher order monadic logic to express pattern solutions. LePUS is based on abstractions of design elements, such as classes, methods, and code and it also includes a visual notation for representing formula of the language. It is strongly based on mathematics and formal logic. They propose a tool, based on Prolog without support for the visual notation. A critic Mapelsden et al. (2002) moved to the framework is that, despite the compact form of the visual notation, it often includes too many different syntactic elements making the diagram difficult to interpret. In a successive work, Mak et al. (2003) propose an extension (ExLePUS) of the initial framework for a better integration with CASE tools also discussing the problem of compound patterns. Patterns contain slots that are filled by other patterns to produce an interconnected architecture.

Mikkonen proposes DisCo Mikkonen (1998), (Distributed Cooperation) which uses a form of Temporal Logic of Actions to formally describe constraint interactions for reactive systems. Therefore, while LePUS focuses on the static aspects of patterns, DisCo is primarily concerned with behavioral aspects. The framework allows for managing interactions among objects whose correctness is ensured by property-preserving refinements.

Both LePUS and DisCo (and their available extensions) greatly reduce the ambiguity of the pattern solution, but they must be complemented with the traditional documentation for what concerns the other aspects of pattern description. Another note is that the proposed level of formalization requires special skills to interpret formula.

In addition, many semi-formal approaches exist in literature, most of which are based on UML Kim et al. (2003); France et al. (2004); Sunyé et al. (2000); Mak et al. (2004); Dong (2002) so to be easily