Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Semi-automatic architectural pattern identification and documentation using architectural primitives

Thomas Haitzer*, Uwe Zdun

Software Architecture Research Group, University of Vienna, Vienna, Austria

ARTICLE INFO

Article history:
Received 8 January 2014
Revised 19 December 2014
Accepted 20 December 2014
Available online 27 December 2014

Keywords:
Software architecture
Architectural component views
Architectural pattern

ABSTRACT

In this article, we propose an interactive approach for the semi-automatic identification and documentation of architectural patterns based on a domain-specific language. To address the rich concepts and variations of patterns, we firstly propose to support pattern description through architectural primitives. These are primitive abstractions at the architectural level that can be found in realizations of multiple patterns, and they can be leveraged by software architects for pattern annotation during software architecture documentation or reconstruction. Secondly, using these annotations, our approach automatically suggests possible pattern instances based on a reusable catalog of patterns and their variants. Once a pattern instance has been documented, the annotated component models and the source code get automatically checked for consistency and traceability links are automatically generated. To study the practical applicability and performance of our approach, we have conducted three case studies for existing, non-trivial open source systems.

1. Introduction

During maintenance and evolution of a software system, a deep understanding of the system's architecture is essential. This knowledge about a system's architecture tends to erode over time (Jansen et al., 2007) or even get lost. In a recent study Rost et al. (2013) found that architecture documentation is frequently outdated, updated only with strong delays, and inconsistent in detail and form. They also found that developers prefer interactive (navigable) documentation compared to static documents. This also reflects our personal experiences as well as those of others. For instance, our colleague Neil Harrison shared the following story from his experiences with large-scale industrial systems (shortened): "Once upon a time I worked on a large system that was already a few years old. It had a well-defined architecture. When I started, I was given copies of three or four documents that described the architecture. In addition, I watched several videotapes in which the architects described the architecture. As a result, I gained a good understanding of the architecture of the system. After a few years, I left the project to work on other things. But several years later I returned. The system was still being used and was under active development. Of course, it had changed greatly to add new capabilities and support changes in technology. Underneath it all, the original architecture was largely intact, but it was much more obscure. I wanted to refresh my architectural memory, so I asked around for the original memos and videotapes. Nobody had even heard of them. Critical architectural knowledge had been lost. People were actually afraid to change the original code, because they did not understand how it worked."

Software architecture documentation or, in case of lost architectural knowledge, software architecture reconstruction (Ducasse and Pollet, 2009) techniques can be used to (re)establish the proper architectural documentation of the software system. An essential part of today's architectural knowledge is information about the patterns used in a system's architecture. Patterns can be seen as building blocks for the composition of a system's architecture (Beck and Johnson, 1994; Buschmann et al., 1996). This is especially valid for architectural patterns or styles which describe a system's fundamental structure and behavior (Lange and Nakamura, 1995). A considerable number of software architecture reconstruction approaches support software pattern identification (Beck and Johnson, 1994; Bergenti and Poggi, 2000; Shull et al., 1996). Most of these approaches (see e.g. Bergenti and Poggi, 2000; Heuzeroth et al., 2003; Krämer and Prechelt, 1996; Philippow et al., 2003) focus on automatically detecting design patterns in the source code. Such pattern identification approaches are often restricted to design patterns that were identified by Gamma et al. (1995) (GoF patterns). Architectural patterns, in contrast, convey broader information about a system's architecture as they usually are described at a larger scale than GoF patterns.

There are a number of important problems in automatic pattern identification in general and especially in architectural pattern

* Corresponding author. Tel.: +43 1 4277 78521; fax: +43 1 4277 8 78521.
E-mail addresses: thomas.haitzer@univie.ac.at (T. Haitzer), uwe.zdun@univie.ac.at (U. Zdun).
URL: http://informatik.univie.ac.at/thomas.haitzer (T. Haitzer), http://informatik.univie.ac.at/uwe.zdun (U. Zdun)

http://dx.doi.org/10.1016/j.jss.2014.12.042
0164-1212/© 2015 Elsevier Inc. All rights reserved.

identification. Existing approaches often only focus on the task of identifying a system's design patterns while the documentation of the reconstructed patterns and the future evolution of the system are not considered (which is just as essential as identifying an architectural pattern).

In addition, architectural patterns are often much harder to detect directly in the source code than GoF design patterns as there is often a large number of classes involved in the implementation of the pattern and the variations between different instances of the patterns are very large. As a consequence of the large number of involved classes there is a possibly huge search space for these patterns that grows with every class and increases execution times (Ducasse and Pollet, 2009).

A big problem of pattern identification is the variability in pattern implementations. Only a very few pattern identification approaches consider pattern variations at all, and they are usually focused on GoF design patterns only (Wendehals, 2003; Wendehals et al., 2001). For instance, hardly any implementation of a system strictly adheres to the Layers pattern (Buschmann et al., 1996) as described in the textbook, but a huge number of systems are designed based on Layers. To give a concrete example, in the definition of the Layers pattern, a layer only has access to the functionality provided by the layer below it. However, this rule is often violated for cross-cutting concerns like performance, security, or logging. As a consequence, many layered architectures contain parts that do not strictly adhere to the Layers pattern. In addition to this, the Layers pattern suggests but does not in any way enforce clean interfaces between the layers. For these reasons, it is hard to automatically detect architectural patterns like Layers.

Another problem of automatic pattern identification is the accuracy of the approaches, which is often not sufficient. That is, some approaches treat pattern instances they find as candidates (Wendehals, 2003). However the likelihood of false positives increases with system size and can lead to precision values around 40% (Krämer and Prechelt, 1996) which means that 60% of the found pattern instances are false positives. This requires substantial manual effort to review the found pattern instances.

In the light of the aforementioned problems, we formulated the following research questions:

RQ1 How far can a semi-automatic architectural pattern approach go toward the goal of identifying the patterns in architectural reconstruction?

RQ2 How far can a semi-automatic architectural pattern approach go toward the goal of maintaining documented patterns during the further evolution of a reconstructed architecture?

RQ3 In how far are the concepts and tools applicable in existing real-life systems?

RQ4 How efficient are the actual pattern instance matching algorithms that are based on primitives?

RQ5 Are primitives and an adaptable pattern catalog adequate means to handle the variability inherent to architectural patterns?

The main contributions of this article are, first, to suggest a novel semi-automatic architectural pattern identification approach that tackles the aforementioned problems that arise during the documentation and evolution of architectural patterns like the variability inherent to patterns, consistency between the documented architecture and the source code, and the large number of source code artifacts that are related to the implementation of architectural patterns. Second, we show the approach's feasibility in terms of tool support (in the context of three open source case studies), and to study the performance of the approach (also in the context of these cases). We aim to assist the software architect during the reconstruction of architectural knowledge as well as supporting the architect in the documentation of the reconstructed architectural knowledge. After the architectural knowledge has been reconstructed and documented with our approach, we support the software architect in keeping the created architectural documentation in sync with the source code of the application. As Clements et al. (2002) state, a strong architecture is only useful if it is properly documented in order to allow others to quickly find information about it.

Our proposed solution is an interactive approach for the semi-automatic identification and documentation of architectural patterns based on a set of Domain Specific Languages (DSLs). It consists of the following main components:

- *Architecture Abstraction DSL:* In our main DSL, the *Architecture Abstraction DSL*, the software engineers can semi-automatically create an abstraction of an architectural component view based on design models or during architecture reconstruction. To address the rich concepts and variations of patterns, we propose to use architectural primitives (Zdun and Avgeriou, 2005) that can be leveraged by software engineers for pattern annotation during software architecture documentation and reconstruction. Architectural primitives are primitive abstractions at the architectural level (i.e. defined for components, connectors, and other architectural abstractions[1]) that can be found in realizations of multiple patterns.
- *Pattern Instance Documentation Tool:* Using the architectural primitive annotations, our approach provides a *Pattern Instance Documentation Tool* which automatically suggests possible pattern instances based on the architectural component view of a system and a pattern catalog.
- *Pattern Catalog DSL:* The pattern catalog contains templates of the architectural patterns to be identified. It is customizable, reusable and integrates support for pattern variability. Our approach leads to a reduced search space for patterns, as we search for patterns only in the created architectural component view instead of the source code.
- *Pattern Instance DSL:* Identified pattern instances are documented using the *Pattern Instance DSL* which uses the artifacts defined in the *Architecture Abstraction DSL* and the *Pattern Catalog DSL* to permanently store pattern instance documentations.

We automatically generate traceability links between the architectural abstractions and the source code (more specifically, the automatically generated class models of the source code), the architectural abstractions and the selected pattern instances, and the pattern instances and the pattern catalog. When artifacts are changed, the traceability links are used to automatically check the consistency of all the artifacts. Automated consistency checking aids the software engineers during the incremental architecture documentation process, when new artifacts are identified and documented. For example, the system automatically detects when the pattern catalog is used to customize an existing pattern and these changes cause an existing instance of this pattern to be no longer valid. The consistency checks are used throughout the evolution of the documented system and report any occurring violations within seconds.

This article is structured as follows: In Section 2 we briefly explain architectural patterns and architectural primitives as our background. We give an overview of our approach in Section 3, and present it in detail in Section 4. In Section 5 we present three case studies of open source systems in which we have applied our approach to test its applicability. As it is crucial for our approach that it works smoothly

[1] Today, the component and connector view (or component view for short) of an architecture is a view that is often considered to contain the most significant architectural information (Clements et al., 2002). Taylor et al. (2010) define *components* as architectural entities that encapsulate a subset of a system's functionality and/or data. Each component has an explicitly defined interface that restricts access to the component's functionality and data as well as explicitly defined dependencies on its required execution context. They define a *connector* as an architectural building block that is tasked with effecting and regulating interactions among components.