Software Architecture Discovery for Testability, Performance, and Maintainability of Industrial Systems

Ganesan, D.

2012

document version
Publisher's PDF, also known as Version of record

Summary

In this thesis, we developed a *practically inspired* approach for architectural analysis of implemented industrial systems. As discussed in different chapters of the thesis, we followed the "industry-as-laboratory" approach by working closely with customers. Essentially, we followed to a large extend the principles of the "Action Research" model, where *change* is the success. By working closely with customers, we attempted to identify and solve "real problems" of their interest. We provided evidence that change has happened, in that customers or collaborators have been offered methods, tools, and design lessons that made impact on the quality of software products.

Based on several real-world endeavors, it is our position that software architectural design is a challenging job. Architects need reference material of proven architectural best practices so that they can build systems that are testable, meet performance requirements, are maintainable, and so forth. Our main goal was to come up with a large body of practically relevant architectural knowledge by systematically reverse architecting a pool of industrial systems and derive an array of recommendations so that quality can be designed in the first place. To efficiently and effectively analyze real-world systems, we followed an architecture-centric approach, meaning that we discovered the implemented software architecture and analyzed its testability, performance, and maintainability. Thus, we developed the Architecture Discovery and Analysis Method (ADAM). To this end, we enumerate the list of contributions of the thesis.

1. We proposed a method for analyzing the structural and behavioral constraints of the specified architectural style with respect to its implementation. The core idea of the method is that architectural styles offer vocabularies and constraints on the types of components and connectors, thus, we could derive rules from styles. We formalized styles using Colored Petri nets as the executable formal language. At runtime, we fed the collected runtime events to CP-nets to discover component-connector views, sequence diagrams, and analyze various constraints of styles. For details, the reader is referred to Chapter 2, Chapter 4, and Chapter 5.

2. We proposed a method for analyzing the architecture of unit test code from a maintainability point of view. We discussed architectural decisions that facilitate or impede unit testing. We offered a list of reusable questions to assess the maintainability of unit test code. We showed that these questions are simple, yet effective, to review unit test code. For details, the reader is referred to Chapter 3.

3. We proposed a method for discovering the software architecture from the implementation, and analyzing quality risks, in particular, risks related to performance, testability (with emphasis on unit testing), and maintainability. The core idea of the method is that architectural decisions of implemented systems are inspired and influenced by dependencies to external entities (e.g., COTS, frameworks, programming language libraries), which can be explored systematically for discovering software architectures and quality risks, hidden deep in the source code. Our method helps us in improving our understanding of relationships between software architectures and testing. For details, the reader is referred to Chapter 6 and Chapter 7.

4. We proposed a method for reverse architecting abstract runtime structures from the source code. We showed that our method was used to reason about testability at the architecture-level of safety critical medical device software. For details, the reader is referred to Chapter 7.

5. We proposed a method for analyzing organizational aspects of the system under analysis and their influence on implemented software architectures. Our method helps us in improving our understanding of relationships between software architectures and organizational aspects. For details, the reader is referred to Chapter 8.

6. We proposed generally applicable recommendations for quality by design instead of quality by tests. Based on architectural analysis of several systems, we derived an array of generally applicable recommendations so that quality can be built-in instead of being tested-in. Our recommendations characterize a) how to avoid architectural violations in the first place, b) how to facilitate testing by explicitly addressing testing related issues during the design of software architectures, and c) how to reduce performance risks by leveraging performance-oriented design patterns. For details, the reader is referred to Chapter 9.

7. We developed a suite of reverse engineering tools. These tools contribute to a) data extraction from implementation either statically from the source code or dynamically at runtime, and b) data abstraction and visualization of the collected fine-grained data. For details, the reader is referred to Chapter 10.

8. In Chapter 11, we revisited the research questions, which were formulated in the introduction, and discussed how we addressed them and also listed related open issues that are of interest for future research.

A good painter knows when to stop painting.