

Abstract Testability Patterns

Wanderlei Souza

State of Sao Paulo Institute for Technological Research
Sao Paulo, Brazil
wandi@uol.com.br

Reginaldo Arakaki

University of Sao Paulo, Polytechnic School
Sao Paulo, Brazil
reginaldo.arakaki@poli.usp.br

Abstract—Testability is a software quality characteristic that exposes the degree to which a software artifact facilitates the testing process. Software testing is a technical and economical problem, it is important to help identify patterns that would improve the industry’s software testing capabilities. This position paper proposes five abstract patterns that improve software testability, which serves as a reference for testers and developers to evaluate the testability support for high reliable software.

Keywords: *Software quality; Design patterns; Design for testability; Software testing; Observability; Built-in testing.*

I. INTRODUCTION

The component testability is an important quality characteristic to evaluate the degree to which a software artifact facilitates the testing process. A lower degree of testability results in increased test effort. Depending on the methods used, testing activities account for 25% to 90% of total project effort [1][2][3][4]. Thus, it is important to help identify patterns that would improve the software capabilities of the industry.

Controllability and observability are the key points to testability [5][6]. To test a component it is necessary to control the inputs (controllability) and observe the outputs (observability). Without these key points it is difficult to improve system testability.

Testability patterns proposed in this paper are based on abstract pattern concept described in [7] and correspond to mechanisms or services that increase overall system observability and controllability, these patterns are more general ideas and are not concerned with any specific implementation technique or software development platform. They should not be confused with testability factors or characteristics. Abstract testability patterns correspond to architectural mechanisms, not testability principles.

A software architect uses a distinct pattern collection to design a system. Architectural patterns provide a predefined set of structures, responsibilities, rules and guidelines to organize the relation between system components. A pattern implements a sequence of design decisions to manage certain system quality characteristics. The testability of the architecture was brought up by Nancy Eickelmann and Debra Richardson in [8]. The authors propose that the architectural decisions must be aligned to testing strategies. In this way, the testability of architecture is the combination between architectural patterns and the testing strategy.

II. ABSTRACT TESTABILITY PATTERNS

A. Built-In Self-Testing (BIST)

Problem: Internal components establish connections to external resources (HTTP connections, database systems, remote calls etc.) and do not have a standard interface to validate directly these integrations. The absence of a uniform way to verify all critical integration points after the system deployment process reduces the system testability.

Solution: Built-In Self-Tests (BIST) is a mechanism to self-report the status and health of individual system components. Built-In Self-Tests (BIST) adds standard interfaces to validate core system functionalities and provides many types of validation possibilities. For example, testing the interface between a component and a database system can be accomplished by invoking the BIST to validate the connection and permissions on system tables.

Consequences: Developers must implement a standard test interface in all BIST related classes. In general ways, it is a minimal overhead to development process, but implement a BIST could be more difficult depending on the complexity of the integration under test.

B. Dependency Injection (DI)

Problem: A business component is difficult to test in isolation because it has a direct reference to external dependencies (third-party components, database systems, web services, etc.) and it is not possible to replace the dependency without changing the source code. The main problem roots from the business component creating the external dependencies.

Solution: It is necessary to inject dependencies into a business component, rather than relying on the component to manage the dependencies itself. Dependency injection is a pattern that can be used to improve the software testability by removing the business component responsibility for instantiating its own external dependencies.

Consequences: There is added complexity to the source code and there are more elements to manage on test automation process. Testers must be able to manage mock objects creation and initialization in order to replace the original code dependencies when necessary.

C. Dynamic Component Management Extension

Problem: A test team has different mock components to inject and simulate external component behavior, but they do not have a dynamic way to change these components. To

choose the component concrete implementation dynamically is fundamental to test automation process.

Solution: The system must provide a standard extension to make the system components and services suitable. This extension defines a management architecture to testable components. Using a standard test extension to manage components increases the system testability by making applications more controllable.

Consequences: Dynamic Component Management Extension enables system components management in a test environment. Security barriers or component *undeploy* must be used to avoid undesired test behavior in a production environment.

D. Testability Logger

Problem: Application events and test related data must be logged for testing purposes. This can lead to redundant code.

Solution: Use the Testability Logger to provide centralized control of logging functionality and takes care of how the testable events are classified and logged. Testability Logger increases the system testability by making applications more observable.

Consequences: The Testability Logger operations (disk IO access, message digests, etc.) impact the system performance.

E. Testability Interceptor

Problem: A tester needs to intercept messages between components for the purpose of verifying the internal behaviors. System also must provide possibility to change application behavior in order to inject and simulate faults in a system.

Solution: Testability Interceptor offers a mechanism to enhance the observability and controllability of a software system by letting components monitor and dynamically change their behavior. Testers can observe and modify functionality without changing the internal logic of components. The Interceptor supports runtime system monitoring and control through Dynamic Component Management Extensions pattern, described in (C).

Consequences: A system design can get more complicated and hard to understand since the developer has to implement the intercepting points.

III. RELATED WORK

Binder [6] presents the Built-In Test concept, a well-known technique for hardware testing, in a software context. We use an abstraction of this concept to describe the Built-In Self-Testing pattern.

Dynamic Component Management Extension is an abstraction layer over JMX technology [9]. We use the JMX ideas and capabilities in order to improve system testability.

The Dependency Injection pattern was first described by Fowler [10] as a specific form of Inversion of Control. We

believe that DI can be used to improve the system testability by abstracting the dependencies out of a component.

The Testability Logger is based on Secure Logger pattern idea [11], but with a focus on system testing and evaluation instead of security reasons.

The Testability Interceptor pattern is related to the Interceptor pattern, which allows services to be added transparently and triggered automatically [12].

IV. CONCLUSION

In this paper we have introduced perspectives to improve system testability and propose a new architectural pattern category: testability patterns. Future work includes other ideas to architectural testability patterns and concrete implementations.

A Java based concrete implementation of these abstract patterns has been used to improve the testability of critical Internet systems in the main portal to content & the Internet provider in Brazil.

REFERENCES

- [1] S. Jungmayr, "Reviewing Software Artifacts for Testability", Proc. of EuroSTAR '99, Barcelona, Spain, November 10-12, 1999.
- [2] R. S. Pressman, "Software Engineering: Practitioner's Approach", European 3rd Edition, McGraw-Hill Book Company, Berkshire, England, 1994, pp. 609.
- [3] Tim Koomen and Martin Pol, "Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing", Addison-Wesley, 1999.
- [4] B. Beizer, "Software Testing Techniques", International Thomson Computer Press, Boston, 1990.
- [5] Roy S. Freedman, "Testability of Software Components", IEEE Transactions on Software Engineering, Vol. 17, No. 6, 1991.
- [6] Robert V. Binder, "Design for Testability in Object-Oriented Systems", Communications of the ACM, v.37 n.9, 1994.
- [7] Eduardo B. Fernandez, Hironori Washizaki and Nobukazu Yoshioka, "Abstract Security Patterns", 2nd International workshop on software patterns and quality (SPAQu'08), 2008.
- [8] Nancy S. Eickelmann and Debra J. Richardson, "What makes one software architecture more testable than other?", Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, San Francisco, California, 1996, pp. 65.
- [9] H. Kreger, "Java management extensions for application management", IBM Journal of Research and Development, IBM Corp. Riverton, NJ, 2001.
- [10] Martin Fowler, "Inversion of Control containers and the Dependency Injection pattern", <http://martinfowler.com>, 2004.
- [11] Christopher Steel, Ramesh Nagappan and Ray Lai, "Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management", Prentice Hall PTR, 2006, pp. 577.
- [12] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. "Pattern-Oriented Software Architecture, Vol. 2 - Patterns for Concurrent and Distributed Objects". John Wiley and Sons, Ltd., 2000.