# Improving Architecture Testability with Patterns

Roberta Coelho
Computer Science Department
PUC-Rio, Brazil
55-21-2540-6915
roberta@inf.puc-rio.br

Uirá Kulesza
Computer Science Department
PUC-Rio, Brazil
55-21-2540-6915
uira@inf.puc-rio.br

Arndt von Staa
Computer Science Department
PUC-Rio, Brazil
55-21-2540-6915
arndt@inf.puc-rio.br

## ABSTRACT
There is a critical need for approaches to support software testing. Our research exploits the information described at *Architectural Patterns* to drive the definition of tests. As a result**,** we intend to assist developers in finding relatively shorter and cheaper paths to high dependable software.

## Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Software Testing and Debugging; D.2.11 [**Software Engineering**]: Software Architecture; D.2.13 [**Software Engineering**]: Reusable Software: Reusable models.

## General Terms
Reliability

## Keywords
Architectural Patterns, Test Patterns, Testability.

## 1. INTRODUCTION
Over the last years, the view of testing has evolved, and testing is no longer seen as an activity which starts only after the coding phase is completed. Software testing is now seen as a whole process that permeates the development and maintenance activities. For a few years there has been an increasing interest in exploiting the information described at the Software Architecture level to improve the testing process [1].

A software architecture is a description of the subsystems and components of a software system and the relationships between them [2]. It shows relevant functional and non-functional properties of a software system.

Researchers have used formal architectural specification as the basis to develop testing techniques [1]. Specification based approaches have not been broadly adopted in industry, because they require expertise that is hardly present. Moreover, these approaches do not ensure fault absence since it depends on the quality of the specification which can also contain faults.

The target of a test can vary from a single module to a group of modules (related by purpose, use, behavior or structure), or a whole system. Some examples of *test strategies* which have been widely adopted are the following:

(i) *System Testing***:** execution of adhoc manual tests in the user interface components;

(ii) *Unit Testing***:** definition of unit tests to some system classes which are, in general, randomly chosen, or to every system class (according to Test Driven Development Extreme Programming practice).

(iii) *Integration Testing*: there is often a tendency to build systems using a "big bang" approach, in which all modules are combined at once and tested as a whole.

Although, these test strategies can eventually help in system quality assurance, there are many disadvantages associated to them, such as: (i) manual tests exhibit high cost and effort to be executed; (ii) a great amount of resources and effort can be wasted due to the codification of many unit tests, which will not be effective; (iii) when no integration testing is done, a set of errors can occur during system testing and the correction is difficult because the cause isolation is impaired. Moreover, during these tests, important system functionalities may be forgotten.

Our approach exploits the information described at *Architectural Patterns* to drive the definition of unit[1] and integration tests.

## 2. Testing Approach
Software Patterns have been adopted as a way of gathering the collective experience of skilled software engineers at each software development phase – architecture specification, system analysis, design and implementation. Patterns promote reuse, however, the reuse via patterns are seldom a verbatim reuse. Patterns provide high level guidance rather than detailed implementation instructions [3].

Architectural Patterns are an important vehicle for constructing high-quality software architectures. An Architectural Pattern specifies the structural organization of a software system. It provides a set of predefined subsystems, specifies their responsibilities and includes rules and guidelines for organizing the relationships between them.

Architectural Patterns [2] explicitly consider many non-functional properties, such as: changeability, interoperability, efficiency, reliability, reusability. However, they do not address testability explicitly [2]. Figure 1 presents the IEEE definition of software testability.


*OOPSLA'05,* October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

[1] Unit tests verify the functioning in isolation of individual subprograms or larger components made of related components.

> ***IEEE 610.12***
>
> *Testability. (1) degree to which a system or component facilitates the establishment of a test criteria and the performance of tests to determine whether those criteria have been met. (2) The degree to which a requirement is stated in terms that permit establishment of a test criteria and the performance of tests to determine whether those criteria have been met.*

**Figure 1: IEEE definition for testability.**

Testability has two facets: controllability and observability [6]. To test a component we must be able to control the input (and internal state) and observe its outputs. However, there are many obstacles to controllability and observability, which results from the fact that a component under test is almost always embedded in another system.

The primary concern of our approach is to improve architectural patterns testability though the definition of a corresponding test pattern. A Test Pattern [3] defines the configuration of objects needed to test the interaction between other modules. It intends to guide the construction of a piece of software which tests other software. We use architectural information and our experience in testing as the basis to define test patterns.

We developed a case study [4] in which we defined a test pattern to the Layer Architectural Pattern [2].

## 3. Case Study

Our case study focused on Layered Information Systems. However, the same approach can be applied to any other system structured according the Layer pattern. Figure 2 illustrates the structure of the proposed Test Pattern.

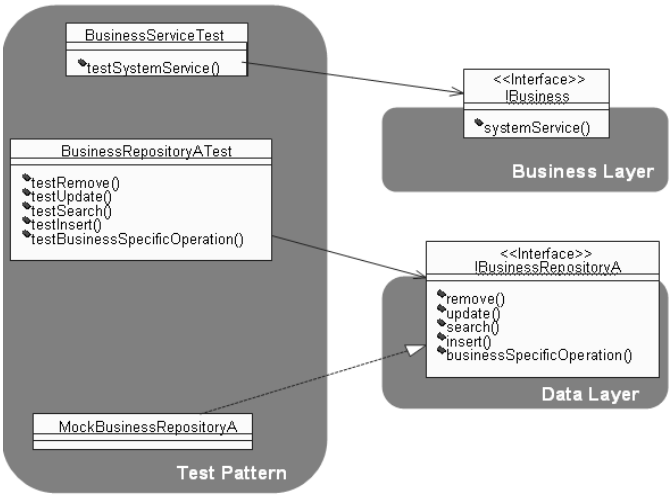


**Figure 2: The structure of Layer Information System Test Pattern.**

The Layered Information System (LIS) Test Pattern has three participants:

- *BusinessTest***:** this class contains all methods that test a set of functionalities provided by the Business Layer Interface and are related according to one specific criterion. This criterion can be a set of operations related to a business entity or to a business service.
- *BusinessRepositoryTest***:** implements test methods to all methods provided by a data access interface. These test classes focus on the testing successful and error conditions of specific data repository functionalities related to insertion, searching, update and other database operations.
- *MockRepository*: this class fakes the implementation of a specific BusinessRepository component. Consequently, this auxiliary class enables the unit test of the business layer.

This pattern guides de definition of unit and integration tests developed to a layered system. It identifies the most important components to be tested, and how they can be tested in unit and integration levels. It allows the execution of three types of tests: Business and Data Layers unit tests, and the test of their integration [4]. This pattern suggests a minimum set of test cases which aims at identifying functional failures. Thus, this proposed set can be extended to follow any of the existing test techniques.

## 4. CONCLUSIONS AND FUTURE WORK

The central point of this approach is to save testing time and effort by recognizing that the test of systems structured according to an architectural pattern can follow a pattern itself. Currently, we have been defining other test patterns to other architectural patterns.

Most software systems, however, cannot be structured according to a single architectural pattern. They must support several system requirements and can only be addressed by a combination of architectural patterns. According to [5] the integration of design patterns shows a synergy that makes the composition more than just the sum of the parts. We plan no enhance our approach by making it also addresses the composition of architectural patterns.

## 5. REFERENCES

[1] A. Bertolino, P. Inverardi, H. Muccini, “An Explorative Journey from Architectural Tests Definition down to Code Execution”, Proc. of International Conference on Software Engineering (ICSE), 2001.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stall, Pattern-Oriented Software Architecture, A system of patterns, Wiley & Sons Ltd., 1996.

[3] J. McGregor, “Test patterns please stand by, Journal of Object-Oriented Programming”, 12(3):14-19, 1999.

[4] R. Coelho, U. Kulesza, A. Staa, C. Lucena, “The Layered Information System Test Pattern”, Fifth Latin American Conference on Patterns Languages of Programming (SugarLoafPlop05), 2005.

[5] D. Riehle, “Composite Design Patterns”, Proc. of OOPSLA 1997, p. 218-228, 1997.

[6] R. V. Binder, “Design for Testability in Object-Oriented Systems”, Commun. ACM 37(9): p. 87-101, 1994.