Testability during Design

Stefan Jungmayr
FernUniversität Hagen, Lehrgebiet Praktische Informatik III
Universitätsstraße 1, 58084 Hagen, Germany
stefan.jungmayr@fernuni-hagen.de

Abstract: Testability is an important characteristic of a software system which has to be considered during all phases and activities of software development. This article describes for object-oriented systems 1) testability issues related to dependencies which are relevant during design and 2) a new approach based on metrics to locate dependencies that are critical for testing.

1 Introduction

Testability is a major factor determining the time and effort needed to test a software system. It is costly to redesign a system during implementation or maintenance in order to overcome a lack of testability. Therefore it is important to deploy testability into the system continually right from the beginning. Relevant testability issues during requirements capture and analysis have been discussed briefly in [3].

There is a large number of software-related factors that effect testability during design. We have grouped them together into a set of nine main testability factors: complexity, separation of concerns, coupling, fault locality, controllability, observability, automatability, built-in-test capability, and diagnostic capability.

In the first part of this article we describe testability issues relevant during software design and focus on coupling caused by dependencies. In the second part we present an approach to define (testability) metrics for dependencies and to locate test-critical dependencies.

2 Testability Issues during Design

2.1 Hard-Wired Dependencies on Classes

During unit test we want to test a class in isolation. Therefore we want to substitute each server class by a stub or mock object [5]. A server class can not be substituted without changing the source code of the client class if a dependency to the server class is hard-wired, for example because the client class creates an instance of the server class using its constructor or because the client class uses static members of the server class.

In order to break hard-wired dependencies 1) make another class responsible to establish the link between the client class and the server class, or 2) let another class establish a link from the client class to an instance of a factory class which allows the client class to get access to an object implementing the type of the server class, or 3) make the client class only aware of the type of the server class and set the link to an instance of the server class at run-time, e.g. by reading a configuration file or by searching a particular directory of the file system.

2.2 Dependencies to Classes

If we want to stub a server class we can implement the stub as a subclass of the server class. This approach is not feasible if the stub should inherit from some other (test framework) class and multiple inheritance is not available, or if the class implements the Singleton pattern [1] and can not be subclassed. To avoid this let every test relevant class implement an interface which is known by its clients.

2.3 Cyclic Dependencies between Classes

Classes within a dependency cycle can not be tested in isolation. Implementing stubs to break dependency cycles causes additional effort. Therefore try to break dependency cycles by extracting interdependent functionality into a common server class ("demotion") or a higher-level class ("escalation") [4], or by using the Observer pattern [1].

2.4 Cyclic Links between Objects

Cyclic links between objects lead to the problem of re-entrance [9]. Maintaining a consistent object state as well as finding and correcting errors becomes difficult in the presence of re-entrance situations. To avoid these problems prefer hierarchical structures of object links, use a polling mechanism to avoid cyclic relationships, or use the Command pattern [1] to reduce coupling in one direction.

2.5 Dense Object Network

Within a dense object network a large number of objects are referenced by more than one other object. This increases the probability of unwanted side-effects which makes it more difficult to locate errors.

Try to avoid dense object networks by calling methods with copies of parameter objects if the original objects shall not be changed. Return new objects from accessor methods [2]. Use immutable objects wherever possible to avoid side-effects in case of shared access [6]. Declare attributes as final wherever appropriate.

3 Dependencies and Testability

Metrics can be used to evaluate the testability of systems and its components. We have adapted the metric ACD from [4] to evaluate overall coupling within a system. ACD, the average component dependency, is the average number of components a system component depends on directly and indirectly.

Local dependencies can have a global effect on testability. Metrics that measure local characteristics of components (like coupling for classes) or global characteristics of an entire system (like the ACD) do not address this phenomenon.

We introduce *reduction metrics* to evaluate the impact of a particular dependency on a particular quality characteristic (like testability). A reduction metric r_m describes the degree to which the value of a quality metric m is reduced if a dependency $d \in D$ is removed (with D being the set of all dependencies within the system).

The value of a reduction metric r_m in percent is defined as follows:

$$r_m(d) = \begin{cases} \left(1 - \dfrac{m(D\backslash\{d\})}{m(D)}\right) \times 100 & \text{if}(m(D) \neq 0) \\ 0 & \text{else} \end{cases}$$

A value of a reduction metric greater than zero in general means that testability improves if the dependency is removed.

As an example we define a reduction metric rACD based on the metric ACD. Figure 1a shows a dependency structure with four components. The value of ACD for this dependency structure is (0 + 0 + 2 + 3) / 4 = 1.25. When we remove dependency x (Figure 1b) the ACD value doesn't change. The value of metric rACD for dependency x is therefore (1 - 1.25 / 1.25) x 100 = 0 percent.

When we remove dependency y instead (Figure 1c) the ACD value decreases from 1.25 to 1.0. The value of metric rACD for dependency y is (1 - 1 / 1.25) x 100 = 20 percent.

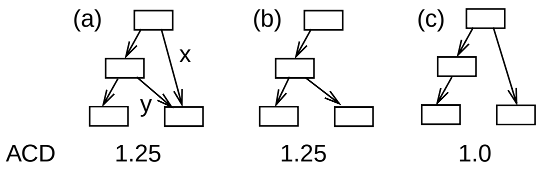

Figure 1: Removing a dependency

Reduction metrics can be used to rank dependencies based on their impact on overall testability. Dependencies that rank high are called test-critical dependencies. In the example above, dependency y is more test-critical than dependency x with respect to metric rACD.

Figure 2 shows the values of metric rACD in decreasing order for the first 10 percent of the dependencies of a system with overall 1853 dependencies and 324 classes. As we can see from this figure, the test-critical dependencies (with respect to reduction metric rACD) are a small fraction of the dependencies with a very high to high impact on the ACD of the entire system.

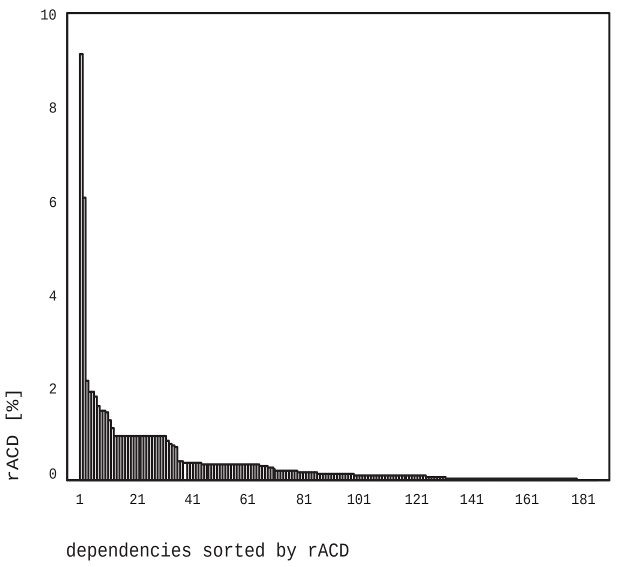

Figure 2: Values of reduction metric rACD

Four case studies have shown that test-critical dependencies are good indicators of design and test problems - many of the most test-critical dependencies could be traced back to a small set of design problems which led to explicit test problems in a number of cases.

4 Summary

There is a large number of testability issues which have to be considered during design. A subset of these issues relate to dependencies. Hard-wired dependencies, cyclic dependencies on class and object level, and dense object networks make testing more difficult. Using design guidelines and reviews to avoid testing problems is one part of the solution to increase testability. Still, local measures may not be able to identify dependencies with an exceedingly negative impact on system structure. The reduction metrics presented in this article are a promising new approach to fill this gap.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*," Addison-Wesley, 1994.

[2] M. Grand, "*Patterns in Java*," Volume 2, John Wiley & Sons, 1999.

[3] S. Jungmayr, "Testbarkeit in den frühen Projektphasen," *Softwaretechnik-Trends*, vol. 21, no. 3, Nov. 2001.

[4] J. Lakos, "*Large-scale C++ software design*," Addison-Wesley, 1996.

[5] J. Link, "Einsatz von Mock-Objekten für den Softwaretest," *JAVA Spektrum*, no. 4, July/August 2001, pp. 53-59.

[6] B. Liskov and J. Guttag, "*Program development in JAVA: abstraction, specification, and object-oriented design*," Addison-Wesley, 2001.

[7] K. Pemmaraju, "Effective test strategies for enterprise-critical applications," *Java Report*, Dec. 1998.

[8] A. J. Riel, "*Object-Oriented Design Heuristics*," Addison-Wesley, 1996.

[9] C. Szyperski, "*Component Software. Beyond Object-Oriented Programming*," Addison-Wesley, 1997.