Abstract

Design for Testability

Stefan Jungmayr

Testability is a software quality characteristic that is of major relevance for test costs and software dependability. Still, testability is not an explicit focus in today's industrial software development projects. Related processes, guidelines, and tools are missing.

This paper is about design for testability, the main intersection of software design and testing. We describe 1) elements of object-oriented software design that may cause testing problems and 2) guidelines on how to improve testability including the use of design patterns. The special focus is on system dependencies and observability.

Keywords

testing, testability, design for testability, design patterns, dependencies, observability

1 Introduction

Testing consumes a significant amount of time and effort within an average software development project. There are different approaches to keep test costs under control and to increase the quality of the product under test:

- improve the software specification and documentation,
- reduce or change functional requirements to ease testing,
- use better test techniques,
- use better test tools,
- improve the test process,
- train people, and
- improve the software design and implementation.

In this paper we focus on software design and how it can be improved to facilitate testing.

The degree to which a software system or component facilitates testing is called *testability*. Designing a software system with testing in mind is called *design for testability*.

There are different ways to improve the testability of a software design, for example to limit the use of class inheritance, to limit the state behavior of objects, to avoid overly complex classes, to avoid non-determinism, and to prepare graphical user interfaces for the use of test automation tools. In this paper we concentrate on system dependencies and observability, describe related testing problems, and give guidelines on how to improve testability. We assume that the reader is somehow familiar with design patterns [Gamm94] and refer mainly to JAVA as an underlying programming language.

2 Dependencies

The static building blocks of object-oriented software systems are classes and interfaces. A class may implement one or more interfaces. Every class and interface defines a *type*. Between a class/interface and another class/interface there may be a *dependency*. Dependencies are necessary for functionality but make testing more difficult. Some types of dependencies are especially problematic for testing and will be described in the following sections, together with guidelines on how to refactor them.

If a class A depends on another class B to fulfill its functionality A is called a *client class* and B is called a *server class*.

2.1 Hard-Wired Dependencies on Classes

Context: A dependency on a server class is *hard-wired* if the server class can not be substituted by some other class (e.g. a subclass of the server class) without changing the source code of the client class. A class A is hard-wired to class B if a method of A

- calls one of the constructors of class B, and/or
- accesses a static method or field of class B.

An example for an access to a static method: Class B implements the Singleton pattern [Gamm94], a method of class A calls the static method `B.getInstance()`.

Problem: During unit test we want to test a class in isolation. Therefore we want to substitute its server classes by stubs or mock objects [Link01]. Hard-wired dependencies to server classes hinder this. Similar problems arise during integration testing if a server class should be stubbed because it is not test-ready.

A hard-wired dependency to a particular class is especially problematic if it manifests in many different code locations.

Solution: There are different ways to break hard-wired dependencies on classes:

- Make another class responsible to establish the link between the client and the server. Add a parameter of the server type to a method of the client class (a constructor, or a method that requires access to a server instance, or a dedicated setup method) which allows other objects to set the link.

  If there are many server classes this approach would result in too many parameters. A variation of this approach is therefore to use a context object [Rain01] (which encapsulates and provides access to a number of server instances) as a parameter.

  During testing the parameter(s) can be used to establish a link to a stub instead.

- Make another class responsible to establish a link from the client to an instance of a factory class [Gamm94] which allows the client to get access to an object implementing the server type. If the server class is a singleton the factory class always returns a reference to the only instance of the singleton.

  During testing the factory class can return a stub instead.

- The client class only knows the type of the server class. The link to one of its instances is set at run-time by reading a configuration file or searching a particular directory of the file system for example.

Assigning the responsibility for object creation and connection of objects to different classes follows the principle "separation of concerns" which in general improves testability.

Consequences: If a client class (is not hard-wired to a server class but) can potentially collaborate with many classes implementing the server type more client-server combinations have to be tested during integration testing. In case of dynamic dependencies the concrete server classes may be unknown at the time of testing which eliminates the confidence usually drawn from integration testing [Szyp97]. Therefore it becomes more important to assure type conformance of the server classes during design and implementation.

2.2 Hard-Wired Dependencies to System Resources

Context: A dependency to a system resource is *hard-wired* if the system resource can not be exchanged without changing the source code of the client class. Hard-wired dependencies to system resources are caused for example by

- explicit declarations of file names in program statements like `new File("C:/dir/filename.ext")` or
- access to standard output/input streams like `System.out.println("...")`.

Problem: Hard-wired dependencies to system resources may hinder testing if these resources are unavailable or not suitable for testing and no other resource can be used instead.

Solution: The solutions are similar to the solutions for hard-wired dependencies on classes. Additionally consider to use a specialized class that handles input/output operations instead of using files or streams. This facilitates logging and the use of mock objects.

2.3 Dependencies to Classes

Problem: If a client class depends on a server class we can implement a stub as a subclass of the server class. Implementing the stub as a subclass of the server class is not possible in case that

- the stub should inherit from some other (test framework) class (but multiple inheritance is not available in Java) or
- the class implements the Singleton pattern. In this case it can not be subclassed because of its private constructor (a super-call to the constructor is not possible) and because of its static `getInstance()` method which can not be overridden (e.g. to return the stub instead of the singleton instance).

Solution: Let every test relevant class implement a corresponding interface. If the class is part of an inheritance tree the root class shall implement an interface or it shall be an abstract class (compare [Souz98], [McGr01]). Each client has only dependencies to the interface that the server implements or to the abstract superclass of the server. (Note: In component based systems remote objects can always be accessed only using their interfaces.)

Consequence: It is not possible to define static methods (like constructors) within interfaces. Therefore it is necessary to use the Factory pattern to create instances [Carm98]. Using a factory class for object creation allows the application to keep track of all instantiated objects. This can be used to monitor resource consumption and to aid debugging during testing and maintenance.

2.4 High Coupling

Problem: The class under test (CUT) depends on a large number of other classes. This makes test tasks more difficult, for example the definition and selection of test cases, the definition of the test order, the creation of stubs, test-setup and integration testing.

Solution: Adhere to the general software engineering principles on how to reduce coupling. Limit the visibility of classes within the system by making them private or protected whenever appropriate. Prefer primitive and language defined parameter types instead of developer defined classes and interfaces. Use the Mediator pattern to reduce the number of classes the CUT has to interact with.

If some basic functionality of the CUT can be tested based on a subset of the server classes define a constructor which refers only to this subset [McGr01].

2.5 Cyclic Dependencies between Classes

Problem: Classes within a dependency cycle can not be tested in isolation. Implementing stubs to break dependency cycles causes additional effort.

Solution: Break a dependency cycle by extracting the interdependent functionality into a common server class (demotion) or a higher-level class (escalation) [Lako96].

Another possibility is to break the dependency cycle by using the Observer pattern [Gamm94] (note: using the Observer pattern does not break the cyclic dependency on the instance level).

2.6 Cyclic Links between Objects

Problem: Objects are coupled via *links*. Cyclic links between objects lead to the problem of *re-entrance*. Re-entrance means, that a method of an object is invoked while another method of the same object is still executing. Maintaining a consistent object state as well as finding and correcting errors becomes difficult in the presence of re-entrance situations [Szyp97].

Solution: Prefer hierarchical structures of object links. A polling mechanism may be a possible solution to avoid cyclic relationships. If cyclic relationships can not be avoided use the Command pattern for class interaction. The Command pattern [Gamm94] decouples method invocation of the client and server class and facilitates logging as well as replay of invocation sequences.

2.7 Dense Object Network

Problem: A tree-like network of object links is preferable in terms of testing. A dense network of object links means that a large number of objects are referenced by more than one other object. This increases the probability of unwanted side-effects. For example:

- A called method may unintentionally change the state of its parameters (Java does not allow to distinguish between call-by-value and call-by-reference parameters).
- Accessor methods may return a reference to an "internal" object that might be changed unintentionally by the calling instance.

Unknown side effects make it more difficult to locate errors.

Solution: Consider the following to avoid dense object networks:

- Call methods with copies of parameter objects if the original objects shall not be changed.
- Return new objects from accessor methods (i.e. methods which must not change the state of its object) [Gran99].
- Use immutable objects wherever possible to avoid side-effects in case of shared access [Lisk01]. Declare attributes as final wherever appropriate.

3 Observability

During and after test execution it is important for testers to be able to observe the return values, intermediate computation results, the sequence of method calls, and the occurrence of faults. The observability of object-oriented systems often needs to be improved.

3.1 Information Hiding

Problem: Private or protected attributes of a CUT can not be accessed by the test driver to set the state of the CUT before test execution and to observe the state after test execution.

Solution: Implement a standard-test interface [Riel96] [Pemm98] within each class including methods like `parse()` and `toString()` to read and write object state and a method `equal()` for comparison of objects.

3.2 Information Loss

Problem: During the test of a method intermediate computation results get lost. From the return value and output parameter values alone it may not be possible to detect a result which is correct by chance despite an error in the method implementation.

Solution: To improve observability of intermediate computation results

- use test-facilities (e.g. assertions) within the method body to report on wrong intermediate results or violations of pre- and postconditions and class invariants,
- implement a test interface that allows a test class to register for internal events of the CUT which carry information on intermediate computation results (and other test relevant information), or
- use a (configurable) logging facility to log intermediate computation results.

3.3 Unobservable Sequence of Method Calls

Problem: The sequence of the method calls and the actual parameter values can not be observed during integration testing without using debuggers or instrumentation.

Solution: Use a pattern that indirects method calls like the Adaptor, Bridge, Facade, Proxy, or Broker pattern. This allows to log the method calls and parameter values. (Note: Proxies can be created during run-time using the dynamic proxy feature available since JAVA 1.3.)

Consequences: Every indirection of method calls causes run-time overhead. Within distributed applications the Broker pattern possibly makes it more difficult to test the collaboration of clients and servers [Busc96].

3.4 Non-Local Faults

Problem: A fault crosses one or more package/subsystem boundaries before it manifests as a failure. It is difficult to locate the error underlying the failure because the failure can not be clearly attributed to the responsible package/subsystem.

Solution: Use a defensive design approach for the package/subsystem interface. Use a filter class to separate the implementation of the interface from the code that shall prevent clients from issuing incorrect method calls. User defined exceptions and runtime exceptions should be catched before they are able to cross package/subsystem boundaries.

4 Summary

In this paper we have presented characteristics of object-oriented design that are critical for testing as well as related guidelines to improve testability. This collection of test problems and testability guidelines can be used as input for company specific programming guidelines, design reviews [Jung99], and the design of application frameworks.

Of course, testability has to be considered throughout the entire software development project. This means to start with testable requirements as well as testability requirements, i.e. requirements related to the testability of the software product. Stakeholders for testability requirements include the customers and software users since testability is important to shorten maintenance cycles and to locate residual errors.

Future empirical studies about the effects and economics of testability are necessary in order to establish testability engineering as a new discipline of software engineering.

5 References

[Arno94] Thomas R. Arnold and William A. Fuson, "Testing in a perfect world," *Communications of the ACM*, vol. 37, pp. 78-86, Sept. 1994.

[Busc96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, "*Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*," John Wiley & Son, 1996.

[Carm98] Andy Carmichael, "Applying analysis patterns in a component architecture," TogetherSoft UK Ltd, 1998, URL http://www.togethersoft.co.uk/.

[Gamm94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software*," Addison-Wesley, 1994.

[Gran99] Mark Grand, "*Patterns in Java*," Volume 2, John Wiley & Sons, 1999.

[Jung99] Stefan Jungmayr, "Reviewing software artifacts for testability," presented at *EuroSTAR99*, Barcelona, Spain, Nov. 8th-12th, 1999.

[Lako96] John Lakos, "*Large-scale C++ software design*," Addison-Wesley, 1996.

[Lind95] F. van der Linden and J. Mueller, "Creating architectures with building blocks," *IEEE Software*, pp. 51-60, Nov. 1995.

[Link01] Johannes Link, "Einsatz von Mock-Objekten für den Softwaretest," *JAVA Spektrum*, no. 4, July/August 2001, pp. 53-59.

[Lisk01] Barbara Liskov and John Guttag, "*Program development in JAVA: abstraction, specification, and object-oriented design*," Addison-Wesley, 2001.

[McGr01] John D. McGregor and David A. Sykes, "*Practical Guide to Testing Object-Oriented Software*," Addison-Wesley, 2001.

[Pemm98] Kamesh Pemmaraju, "Effective test strategies for enterprise-critical applications," *Java Report*, Dec. 1998.

[Rain01] J.B. Rainsberger, "Use your singletons wisely," *IBM developerWorks*, July 2001, URL http://www.ibm.com/developerworks/.

[Riel96] Arthur J. Riel, "*Object-Oriented Design Heuristics*," Addison-Wesley, 1996

[Souz98] Desmond D'Souza and Alan Wills, "*Catalysis*," 1998.

[Szyp97] Clemens Szyperski, "Component Software. Beyond Object-Oriented Programming," Addison-Wesley, 1997.