Testability and Unit Testing

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

Abstract: A prerequisite for unit testing is the possibility to test the unit under test in isolation. Ignoring this requirement during system design and implementation can severely decrease the testability of a software system. This article describes design guidelines and metrics which support software developers in avoiding testability problems.

1 Introduction

Testability got recent attention in the context of Test-First Design[1] which is an important element of Xtreme-Programming [1] [5]. Test-First Design means to write test cases before actually coding the body of member operations. By doing so software developers get early feedback about the impact of dependencies on the testability of the code and hints on which dependencies should be refactored. While such an approach might be sufficient within an Xtreme-Programming context, large projects following a conventional development process require a more systematic approach to testability.

In this paper we focus on dependencies within object-oriented systems and their effect on the testability of the system implementation. First we describe dependencies and their effect on unit testing, then we give some guidelines on how to design for unit testability, and present metrics that can be used to evaluate unit testability.

2 Dependencies

In the context of testing we focus on syntactical dependencies between classes[2]. A *syntactical dependency* from a class A to a class B means that class A needs class B in order to compile correctly. The dependent class is called *client* (class), the dependee class is called *server* (class). Dependencies define a relation between classes, i.e. between two classes there is at most one direct dependency in each direction.

We distinguish between dependencies to types and hard-wired dependencies:

1) A *dependency on a type* means that the server instance can belong to any class implementing the type of the server class. The type of the server class can be specified by 1a) an interface, 1b) an abstract class, or 1c) a concrete class.

2) A *hard-wired dependency* means that the server instance has to belong to a concrete class. The call of a class constructor for example always leads to a hard-wired dependency.

3 Unit Testing and Dependencies

Unit testing means to focus on a small part of a software system. During unit testing it is much easier to isolate faults compared to system testing.

A frequent situation encountered during unit-testing is that the class-under-test (CUT) depends on other classes. A common approach is to substitute these server classes by stub or mock objects [4] which has the following advantages:

- it avoids problems with server classes not ready for testing,
- test set-up is easier,
- faster testing is possible (by removing dependencies to classes involved in user interaction, access to data bases, internet protocols, etc.), and
- failures are detected earlier (when using mock objects).

In order to be able to substitute the server class by a stub or mock object 1) the stub or mock object has to implement the type of the server class and 2) the CUT has to use a reference to the stub or mock object instead of a reference to an instance of the server class.

The category of the dependency to the server class (as defined in Chapter 2) is important for our ability to test a class in isolation:

Category 1a: The stub or mock object can belong to any class implementing the type specified by the interface which means highest flexibility.

Category 1b and 1c: The stub or mock object must be an instance of a subclass of the server class (and therefore it can not inherit at the same time e.g. from a test framework class if multiple inheritance is not available).

Category 2: It is not possible to use stubs or mock objects instead of instances of the server class without changing the code of the client class.

4 Design for Unit Testability

We propose the following strategy to achieve unit testability with respect to dependencies:

- Identify each class c_{ut} to be unit tested, e.g. business entity classes or classes implementing critical functionality.

1 Also called Test-Driven Development.
2 By classes we also mean interfaces as long as we do not distinguish them explicitly.

- Design and use factory classes [2] for each c_{ut} to separate object creation from object usage. If there is only one instance of c_{ut}, then let the factory class handle this.
- Avoid hard-wired dependencies from c_{ut} to its server classes, i.e. avoid direct access to static members of the server classes.

  Possible exceptions to this rule are for example instances of exception classes which should be created directly without involving a factory class.

5 Improving Unit Testability

To improve the unit testability of an existing implementation we try to identify those hard-wired class dependencies which are easy to be refactored into dependencies on types.

In this context we analyze *dependencies on the statement level* contributing to class dependencies. A dependency on statement level is caused by a method call, a type declaration of a parameter, or a statement declaring an inheritance relationship for example. If there is at least one hard-wired dependency on the statement level the respective class dependency is hard-wired as well.

A hard-wired class dependency, which is caused only by a small number of hard-wired dependencies at the statement level is a good candidate for refactoring into a dependency on a type.

6 Unit Testability Metrics

In order to maintain and improve the unit testability of a system we want to 1) evaluate the overall system w.r.t. hard-wired dependencies, and 2) to identify problematic dependencies or dependencies with improvement potential. The following metrics help to do so:

m1 *percentage of hard-wired class dependencies*

This (sub-)system-level metric helps to evaluate the degree to which a (sub-)system suffers from hard-wired dependencies.

m2 *percentage of classes without hard-wired dependencies*

This (sub-)system-level metric (in combination with metric m1) allows to evaluate the degree to which hard-wired dependencies are spread over all classes of the (sub-)system.

m3 *average number of transitive hard-wired class dependencies*

This (sub-)system-level metric helps to evaluate the average degree to which a class depends on other classes because of hard-wired dependencies.

m4 *percentage of hard-wired dependencies on statement level*

This metric, applied to individual dependencies, indicates hard-wired dependencies with a probably low associated effort to transform them into a dependency on a type.

7 Case Study

We studied a software system consisting of 273 classes and interfaces. Not considering GUI-related classes, exception classes, and interfaces there are 155 test-relevant classes.

The software system is very difficult to test because of a huge number of hard-wired dependencies to classes including those involved in data base access as well as cyclic dependencies [3]. Using a prototype metric tool able to collect data concerning the metrics described in the previous chapter the findings are:

- Overall there are 1085 direct class dependencies of the test-relevant classes on other classes and interfaces. About half of these direct class dependencies (536) are hard-wired (m1 = 49%).
  - 12 of these hard-wired dependencies are caused by a combination of inheritance and static access which should be refactored.
  - 441 of these hard-wired dependencies are caused by static access without involving inheritance. 149 of these 441 dependencies (34%) involve object creation. This means that object creation and object use are not well separated in many cases.
- The percentage of test-relevant classes without any hard wired dependency is low (m2 = 14%).
- On average, each test-relevant class is hard-wired to 51 other classes in a transitive manner (m3 = 51.48).
- 95 (out of 536) hard-wired class dependencies are caused by less than or equal to 25% hard-wired dependencies at the statement level ($m4 \leq 25\%$) highlighting first candidates for refactoring.

8 Summary

Hard-wired class dependencies can have a serious impact on the ease of unit testing. The design guidelines and metrics described in this article can help development and test teams to avoid related test problems.

Acknowledgements

Many thanks to Edgar Merl for implementing the code analysis unit of the prototype metric tool and for investigating test problems during the case study.

References

[1] Kent Beck, "Test-Driven Development by Example", Addison Wesley, 2003.

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

[3] S. Jungmayr, "Identifying test-critical dependencies," in Proceedings of *IEEE International Conference on Software Maintenance*, Montréal, Canada, 3-6 October, 2002, pp. 404 - 413.

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

[5] J. Link, "Unit Tests mit Java: Der Test-First-Ansatz", dpunkt.verlag, 2002.