Reviewing Software Artifacts for Testability

Stefan Jungmayr

FernUniversität Hagen, Praktische Informatik III
Feithstrasse 142, D-58084 Hagen, Germany
stefan.jungmayr@fernuni-hagen.de

Summary Testability is the degree to which a software artifact facilitates testing in a given test context. It is directly related to test effort reduction. A lack of testability, like other design faults, is expensive to repair when detected late during software development. Therefore, testability should be addressed already during reviews of early development artifacts. This paper describes different aspects of testability, heuristics that help to evaluate the testability of a software artifact, and how reviews based on checklists can be used to control the deployment of testability throughout the software lifecycle.

1 Introduction

Test costs

Software testing is an expensive activity during software development. About 40 to 50 percent of the overall development budget is spent on testing. Resources are needed during the activities of test design, test preparation, test execution, and test result analysis. (Other costs closely related to testing are costs of fault isolation and costs of fault removal but are out of scope).

Less costs by improved test criteria

The improvement of test criteria is one way to reduce test costs. Better test criteria help to create better test samples, i.e. to reduce the number of test cases needed to achieve a particular fault detection ratio.

Less costs by test automation

Test automation is a way to reduce the effort associated with recurring test activities with help of test tools.

Less costs by automated test oracles

The costs of analyzing the test results are linked to the oracle problem of testing: we need some authority that is able to distinguish between correct or false test results. Humans are expensive test oracles. The implementation of an automated test oracle is an approach to reduce the costs of test result analysis.

Less costs by testability engineering

Certain characteristics of a software system make it easier or harder to test it and to analyze the test results. Testability comprises these characteristics and is an important factor to achieve an efficient and effective test process. We call the systematic approach to high testability of software systems *testability engineering*.

Testability engineering is somehow orthogonal to the approaches mentioned before. By testability engineering we want to make the definition of effective test coverage criteria as well as test automation easier, and to tackle the oracle problem.

The left part of Table 1 summarizes how approaches to test cost reduction influence the costs of test activities (a '+' denotes an influence, a '++' a strong influence).

Recurring and nonrecurring test costs

The resources spent on the different test activities differ for initial testing, re-testing during development, and regression testing during software maintenance. The effort for test design, for example, will be higher during initial testing than during regression testing. Cost reductions for initial testing can be gained once (nonrecurring benefits), cost reductions for regression testing and re-testing can be gained multiple times (recurring benefits).

The right part of Table 1 shows the relationship between cost reductions of particular test activities and recurring savings for initial testing, re-testing, and regression testing.

	Approaches to Test Cost Reduction				**Recurring Savings**		
Test Costs	test criteria	test automation	automated oracle	testability engineering	initial testing	re-test during development	regression testing
test case design	+			+	++	+	
test execution	+	++		+	+	+	+
test result analysis		+	++	+	+	+	++
fault isolation				+	+		
fault removal				+	+		

Table 1: Approaches to test cost reduction.

Testability Reviews

Testability reviews are one technique of testability engineering and can be applied throughout all software development phases. They allow test engineers to evaluate the testability of software artifacts early and to trigger necessary design changes when it is still relatively inexpensive.

Types of testability reviews

Testability can be checked as part of reviews with a broader perspective or as pure testability reviews with testability considerations as their main focus. The rigidity of the review process can range from informal to formal. For the sake of simplicity we will refer to all these reviews as testability reviews if they focus on testability issues to some extent.

Paper outline

In this paper we will first have a short look at related work in the field of hardware testability and software testability (Chapter 2). Then we will make our perception of testability more precise and look at different aspects of testability (Chapter 3). After that we describe a selection of

testability heuristics (Chapter 4), guidelines on how to derive a checklist based on these heuristics (Chapter 5), and a general strategy for testability engineering (Chapter 6). We close our discussion of testability reviews with a summary and hints on other online-resources (Chapter 7).

2 Related Work

There has been a lot of work on testability in the hardware field and some in the field of software. This chapter shortly presents mature approaches.

Hardware Testability

For more than two decades now there is a continuos research effort on the testability of highly integrated circuits. The strong attention given to the topic of testability is obvious from the existence of conferences and standards especially dedicated to this topic.

Testing problem

Highly integrated circuits combine a large amount of logical components but have only a limited number of input and output pins. This causes problems for hardware testers: it is difficult to control single logical elements on the circuit and to observe their behavior because their inputs and outputs have to pass many intermediate logic elements.

Approaches to test cost reduction

Hardware testers tackle these test problems by increasing the controllability and observability of integrated circuits. The approaches include boundary-scan architectures — on-circuit test facilities that allow to by-pass intermediate logic elements and directly control particular circuit elements.

Similarities to software testing

The problems in the software field are somehow similar with respect to the small number of component input and outputs available for testing compared to the complex internal logic to be tested. Controllability and observability in the context of software testability have already been discussed for example in [Free91] [Bind94] [Gupt94].

Differences to software testing

The difference to software testing is that hardware testing primarily addresses production defects and not design faults. *Each* logical component of a circuit has to be checked if it works correctly. In software we don't have to check whether logic components (like IF- or WHILE statements) work correctly - in software we have to check if the statements altogether suit the intended functionality. Of course there are design faults in chips as well which are not the focus of hardware testability (but of simulation runs).

Testability of Distributed Real-Time Systems

Challenges of distribution

Internal system behavior of distributed real-time systems is difficult to observe and monitoring test results is a major challenge. The observation itself may cause undesired effects on the timing behavior of the system (probe effect). Another challenge is the non-reproducible behavior of distributed systems. If test execution results are not repeatable they can not be used during regression testing.

Architecture to match challenges

A key approach to master this challenges is an appropriate system architecture. Time-triggered systems, for example, make it much easier to achieve reproducible test runs and to avoid probe effects then event-triggered systems [Schu93].

Testability Assessment by Mutation Analysis

Faults hiding from test cases

The likelihood that faults are hiding from a particular testing scheme has been investigated by Jeffrey Voas [Voas95]. This likelihood is a function of the likelihood that a particular program location is executed, the likelihood of a fault at that location causing a wrong data state, and the likelihood of a wrong data state propagating to the output state. A tool based on the generation of program mutants has been implemented to calculate this likelihood. This approach is able to highlight areas of code which are likely to hide faults and where changes to code statements would be appropriate. The calculation of testability allows to make reliability predictions more accurate, too.

3 Testability Model

In this chapter we want to discuss the concept of testability in more detail.

The definition of testability given by IEEE is:

> *The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.* [IEEE90]

Testability is Context Dependent

There is no absolute measure of quality. Software quality is always defined in terms of particular customer needs. Something similar can be said for testability: Whether a component or system is testable or not depends on the context. The context relevant for testability assessment includes:

- test constraints
- intended use of the component

- test criteria
- test tools

Testability and test constraints

There are two basic constraints in testing - cost and quality. Test *costs* depend on the *efficiency* of the test activities, the *quality* achieved depends on their *effectiveness*. These test constraints have to be taken into account when evaluating testability. The reason for this is that some problems caused by lack of testability primarily effect the test effort, others the fault detection rate of tests based on particular test criteria. The lack of a standard test interface, e.g., increases the test effort but does not necessarily lead to lower test effectivity.

Additional test constraints relevant to testability may be test duration or test automation feasibility (in case of massive regression testing effort).

Testability and intended use

A component to be tested can be intended for a particular use within a system or for reuse within different systems. In the first case the usage context of the component is well defined and can be used as a basis for test case definition. In the second case we have the problem that test results may be invalid if the component is executed within a new environment not known to us beforehand. A test approach differs for different types of intended use and so do the testability requirements.

Testability and test criteria

Depending on the test criteria applied we have to address different issues of design for testability. A reason for this is that test efficiency and test effectivity are less sensible to characteristics of the program code when specification based test criteria are used then if program based test criteria are used.

Testability and test tools

The (non-)availability and capability of test tools (like test tools for distributed systems) often leads to testability requirements, like the implementation of particular test interfaces.

Testability and Development Phases

Software testability is not a characteristic of source code artifacts alone. From the domain of distributed real-time systems we know that the architecture (high-level design) of a software system can be a main factor of testability. We want to extend this view and refer to testability as an characteristic of a software artifact (a system, component, or document) independent of the current abstraction level.

Definition of Testability

Our working definition of testability is:

> The degree to which a software artifact facilitates testing in a given test context.
>
> The test context includes the test constraints (like available budget and required quality), the intended use of the component (e.g. spe-

cial purpose or reuse), the test criteria applied, and the test tools used.

This working definition of testability is similar to the IEEE definition but it emphasizes that testability is a context dependent attribute of software artifacts from different development phases.

Testability and "Good Design"

Testable design is more specific then "good design" (i.e. design that follows acknowledged design principles) because it is explicitly intended to match a particular test context. Aspects of testability like observability and reproducible behavior are not the primary focus of good design and require special treatment.

Sometimes a trade-off may be necessary between testability, performance and other quality characteristics of software.

4 Heuristics for Testability Reviews

Reviews have been adopted in industry as an effective and efficient mean to detect faults [Acke89] [Russ91] [Basi97]. Especially the acceptance of reviews of early life-cycle artifacts is good. Testers often participate in the review meetings which is a recommended practice. These reviews are the right place to consider testability because they allow testers to highlight potential testing problems. In the following we describe examples of testability heuristics that should be checked during testability reviews. The heuristics summarize testability considerations given in the testing literature, experiences from test practitioners, and reflections based on object-oriented testing methodology. Some of the heuristics only apply to object-oriented software, others apply to both object-oriented and conventional software.

Architectural Level

Distributed control structure

Control structure in object-oriented systems is often distributed over several components. To test a function related to this control structure requires the availability of all involved components, to understand all these components, and to instantiate all of them. This is problematic in large projects because delays in one development team may delay the testing of the distributed control structure. Distribution of control structure may be adequate for a cluster of closely related classes, it is not adequate for a large system.

Heuristic: Concentrate control structure related to a particular functionality in one cluster (or class).

Reuse

Favor modularity before reuse. Its better to have code duplicates than to delay testing of a component because required changes to a superclass or library class it depends on are pending.

Heuristic: Give higher priority to the modularity of a system than to the reuse of components.

Standard test interface

A standard test interface in all classes causes minimal additional implementation effort and has a high potential payback. [Bind94]

Heuristic: Implement a standard test interface within each domain class. Implement a standard test interface within base classes and technical classes if they are selected for direct class testing.

Observation points

Observation points are additional interfaces that allow to collect information about intermediate processing or communication steps that can otherwise not (or not easily) be observed. The observation points shall be introduced at locations where test result analysis based on the observation results is simplified. For example, information exchanged in higher level protocols is easier to interpret than the representation of this information in low level protocol formats.

Heuristic: Introduce observation points at semantically meaningful points.

Inheritance and test strategy

Polymorphic method calls between classes within an inheritance hierarchy may require re-testing of the superclass when a subclass changes and vice versa. There are at least three different ways you can deal with this problem:

1. Avoid inheritance, use delegation instead: re-testing is not necessary.
2. Use inheritance, perform an analysis of the dependencies between super- and subclasses, and use a selective regression testing strategy.
3. Use inheritance, don't analyse dependencies and rerun all test cases (test automation strategy).

Heuristic: Map your test strategy and your design approach with respect to inheritance hierarchies.

Design Level

Control structure

Control structure can be hidden in data. Mechanisms like look-up tables represent a programming language of their own. If you make this language explicit you can devise coverage criteria, the chance of omitting important test cases is reduced.

Heuristic: Make control structures explicit.

Cyclic dependencies

When there are cyclic dependencies between classes it is not easy to determine a test order that minimizes the amount of stubs and drivers necessary [Wint98]. In case of cyclic dependencies between methods it is not possible to avoid stubs and drivers by means of a particular test order which increases the test preparation effort significantly.

Heuristic: Avoid cyclic dependencies, especially between methods.

Method overwriting

You can reduce the number of possible polymorphic method calls within an inheritance hierarchy if you declare methods to be final (JAVA) or non-virtual (C++). This reduces the analysis effort if you follow a selective regression testing strategy. Especially for abstract classes within a framework make explicit which methods are intended for overriding and declare other methods as final (non-virtual).

Heuristic: Declare methods as non-virtual if they are not intended to be overridden.

Polymorphic parameters

Polymorphic method parameters or attributes can contain different classes from within an inheritance hierarchy. To exhaustively test a method with polymorphic parameters you have to test all combinations of parameter classes which is often unrealistic. To deal with this problem you can follow different strategies:

- The inheritance hierarchy of the parameter classes realizes strict subtyping — consider only parameter root classes during testing.
- Test all combinations of parameter classes, therefore minimize the number of parameters and restrict the type of the parameter class (i.e. the maximum level within the inheritance hierarchy) by appropriate casts whenever possible.

Heuristic: Avoid unmotivated polymorphic method parameters, especially if strict subtyping does not apply.

Implicit dependencies

It is easy to miss implicit input and side-effects of methods, especially if they are not well documented. Unknown dependencies often lead to unreproducable test results; uncovering those by testing is costly. Test design is easier in case of explicit input and output parameters (as part of a methods signature) and an explicit distinction between side-effect free functions and procedures.

Heuristic: Avoid implicit inputs and side-effects.

Law of Demeter

Any method *m* that obeys the Law of Demeter only knows about the immediate structure of the class *C* to which it is attached. The structure of the arguments and the sub-structure of *C* are hidden from *m* [Lieb88] [Macd95]. Compliance to the Law of Demeter reduces the number of interfaces, the number of stubs and drivers that may be needed, and the number of integration test interfaces. [Bind94]

Heuristic: Follow the Law of Demeter.

State behavior

Objects respond differently to similar method invocations in case of state dependent behavior. Then it is necessary to test each method for each object state. This increases the test effort considerably, especially during integration testing of interacting objects with state behavior.

Heuristic: Avoid unmotivated state behavior of objects.

Encapsulation

The state of an object is an important part of the test result after each test case execution but normally not accessible from the outside. Encapsulation makes testing more difficult. Breaking the encapsula-

tion introduces unwanted dependencies between test drivers and the class to test.

Heuristic: Implement a state testing function for each test relevant class. Prevent the invocation of state testing functions by components other then test drivers if necessary. (As an alternatively, use assertions related to pre- and postconditions to control correct states after state transitions.)

Built-in-test facilities

Intermediate computation results can be relevant for testing but it may not be possible to reconstruct them from component outputs (information loss). Built-in-test facilities like assertions provide a way to control intermediate computation results. (They can be faulty, too, and should be checked by reviews.) [Bind94]

Heuristic: Compensate test relevant information loss by built-in-tests.

Unachievable output values

Output coverage is one goal of test case design. Unachievable output values (called output inconsistencies) may be indicators for unreachable statements and paths [Free91]. Output inconsistencies (which may already be present in the specification or design) shall be avoided to make test result analysis easier.

Heuristic: There has to be at least one input element (or combination of input values) for each output element [Karo96] [Free91] [[Bach97]].

Exceptions

Exception handling is an important feature of software. Some exceptions do never occur according to theory but are considered by exception handling for safety reasons. Since these exceptions should never occur it is difficult to trigger them during testing. A testable exception handling requires a design strategy and perhaps simulation of failure modes. [Bind94]

Heuristic: Provide means to trigger all exceptions.

Code Level

Elegant solutions

Elegant solutions and performance tweaks tend to reduce the correspondence of the implementation and its specification. Understanding the code and designing test cases to achieve code coverage is more difficult, observation and interpretation of intermediate computation results is more difficult as well. [Beiz90] [Bind94]

Heuristic: Don’t squeeze the code.

Variable reuse

Variables are reused if there value is defined more than once in a method body. Variable reuse leads to implicit information loss, i.e. loss of intermediate computation results and can be avoided by using more variables. [Voas95]

Heuristic: Avoid variable reuse.

Unachievable paths

Unachievable paths lead to problems during coverage based testing and may be difficult to determine. The total number of paths, the total number of achievable paths, and the number of paths needed to achieve path coverage should be as close to each other as possible

[Beiz90]. In order to reduce the number of unachievable paths avoid correlated decisions. [Beiz90]

Heuristic: Minimize the number of unachievable paths.

Recursion

It is difficult to test recursive algorithms because we can not easily create a stub for the component or method we want to test. (A solution to this problem is to split it into pairs that call each other [Lidd96] or to use built-in assertions.) Note: unanticipated recursion can occur in object-oriented code because of inheritance and self-reference.

Heuristic: Avoid recursive implementations of algorithms if there is no checking of invariants.

5 From Testability Heuristics to Review Checklists

Checklists are dynamic

Review checklists are used to aid reviewers during their individual reading (preparation phase). They shall guide the reviewer to detect common faults that are important with respect to the focus of the review. Checklists are sooner or later out of date if they are once written and never changed again. Items on a checklist change as the related types of faults are detected less frequently and other types of faults become more relevant. Therefore it is necessary to regularly update checklists. (Note: still keep track of old checklist versions.) For a checklist to be practical it should not exceed one page and 20 to 25 items.

Checklist items

The testability heuristics described in the previous chapter can become items of your own testability checklist. Another important source of checklist items with respect to testability are the people of your organization involved in testing. At the end of a development project it is important to discuss difficulties during testing and how changes in the attitude to architecture, design, and code could help testers.

Checklist structure

The order of the checklist items should correspond to the importance of the related testing problems. Give a reference to company standards or guidelines if applicable. Provide the reviewers with background information (i.e. motivation) for each checklist item, for example, on the flip-side of a paper-based checklist or as a hyperlink to follow in case of an online-checklist.

Checklist types

According to the type of reviews you perform set up corresponding types of checklists.

Fault tracking system

If you have installed a customizeable fault tracking system add a fault type related to a lack of testability. This allows to track the sources of testing problems.

Review form

A reviewer shall note in the review form (used to collect faults) which checklist item led to the detection of a fault if applicable. This allows to analyze the usefulness of different checklist-items.

Checklist versus standards

Company standards and guidelines are another way to assure certain quality characteristics of software. Standards and guidelines are used as reference documents during reviews, too. What makes them different from checklists is that they are not related to the frequency of the most common faults currently observed during reviews. It is appropriate to take into account testability heuristics within company standards, too. Note: Testability heuristics that are covered by company standards shall become part of testability checklists if the company standard is often violated in this respect.

Checklists and automated checking

Of course, every testability heuristic that can be checked automatically by means of static analysis tools should be checked in this way before carrying out testability reviews.

6 Testability Engineering Strategy

The evaluation and design of software artifacts for testability consumes resources and shall follow a strategy. In this chapter we outline a strategy to optimize the return on investment of testability engineering.

Evaluate testability of system architecture

First, evaluate the system architecture for testability and enhance testability where necessary and possible.

Define testability requirements

Second, identify the components of your system that are critical for testing. If a risk assessment is part of your project planning use these results. In general, a component is assigned a high risk with respect to testing success if

- it realizes critical functionality and/or has a high usage frequency: in this case poor testing leads to poor product quality;
- it lies on the time critical path: test problems may propagate and prevent testing of other components;
- is tested often (regression testing): existing testing problems are multiplied by each regression test run.

Based on the risk assigned to a component define a required testability level.

Describe test context

Third, describe the test context for each component with high testability requirements. Start with describing the test constraints - it must be clear, whether the focus for testability is on test effectivity or test efficiency. Identify the intended component use, the component test strategy, test criteria and test tools to be applied as well as available documentation. This task is normally covered by test planning.

Perform testability reviews

Fourth, perform a testability review for each component with high testability requirements.

Define required design changes

Fifth, identify testability faults and define changes of the component design and/or implementation if necessary.

Collect experience

After finishing a major release of a software compare the test problems reported for larger components with the effort invested in test-

ability engineering beforehand. Reflect the lessons learned from this analysis in a revised version of the testability checklist.

General test strategy and standards

By making selected testability considerations part of your development standards you can free testability reviews from the related burden. For example, if your company emphasizes test automation, then you may want to make requirements on standard test interfaces part of the company standards and guidelines.

7 Summary

Dealing with testability issues during reviews of software artifacts is a first choice mean to achieve an effective and efficient test process. In this paper we emphasized that testability is not a characteristic of source code alone but applies to the architecture and design level as well. We emphasized the concept of a context dependent testability, not only dependent on the test criteria applied but also dependent on the test constraints, intended use and test tools. We described checklists as a way to focus on the most important testability issues during reviews. The testability heuristics presented provide a starting point for defining your company or project specific checklists.

Core concepts of object-oriented technology like polymorphism and late binding are means to build flexible, extendable, and reusable software applications. If we don't want to suffer from testing problems we have to control the use of these powerful concepts by testability engineering.

Future work has to investigate testability economics, that means the costs caused by testability engineering and its short- and long-term effects on test efficiency and effectivity.

Resources

At http://www.testability.de/ you can find more material on software testability including a bibliography on software testability, and links to other online resources related to software testability.

Acknowledgments

Many thanks to Henrik Behrens, Peter Pauen, Mario Winter (all FernUniversität Hagen), Gerald Futschek (Technical University Vienna), Franz Mauerer (START), Sven Nordhoff (DTK), and Steffen Weisbrod (Systor) for their reviews on draft versions of this paper.

8 References

[Acke89] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software Inspections: An Effective Verification Process. *IEEE Software*, May 1989, pp. 31--36.

[Bach97] J. Bach. *Attributes of Software Testability.* 1997. http://www.stlabs.com.

[Basi97] V. R. Basili. Evolving and Packaging Reading Technologies. *Journal of Systems and Software*, vol. 38, pp. 3--12, 1997.

[Beiz90] B. Beizer. *Software Testing Techniques*. 2nd ed. International Thomson Computer Society Press, 1990, ISBN 1850328803.

[Bind94] R. V. Binder. Design for Testability in Object-Oriented Systems. *Communications of the ACM*. September 1994, pp. 87--101.

[Free91] Roy S Freedman. Testability of Software Components. *Transactions on Software Engineering*. June 1991, pp. 533--564.

[Gupt94] S. C. Gupta and M. K. Sinha. Improving Software Testability by Observability and Controllability Measures. In Proceedings of the IFIP 13th World Computer Congress, Hamburg, Germany, 28 August - 2 September, 1994, pp. 147--154. North-Holland, 1994, ISBN 0-444-81989-4.

[IEEE90] IEEE Std 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*. 1990.

[Karo96] K. Karoui and R. Dssouli. *Testability analysis of the communication protocols modeled by relations*. Technical Report TR 1050, Universite de Montreal, Department d'Informatique et de Recherche Operationelle. November 1996.

[Lieb88] K. Lieberherr and I. Holland and A. Riel. *Object-Oriented Programming: An Objective Sense of Style*. In Proceedings of OOPSLA '88. 1988.

[Lidd96] J. Liddiard. Is your software designed to be testable? In *Testing Times, The IPL Software Product Newsletter*. October 1996, pp. 2-3,6.

[Macd95] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Applying inspections to object-oriented software. Technical Report RR-95-188, University of Strathclyde, Department of Computer Science, June 1995.

[Russ91] G. W. Russell. Experience with Inspection in Ultralarge-Scale Developments. *IEEE Software*, January 1991.

[Schu93] W. Schütz. The testability of distributed real-time systems. Kluwer Academic Publishers, 1993, ISBN 0-7923-9386-4.

[Voas95] J. M. Voas and K. W. Miller. Software Testability: The New Verification. *IEEE Software*, vol. 12, no. 3, May 1995, pages 17-28.

[Wint98] M. Winter. *Managing object-oriented integration and regression testing (without becoming drowned)*. In Proceedings of EuroSTAR 98. 1998.