Exploring Languages with Interpreters and Functional Programming Chapter 11

H. Conrad Cunningham

24 January 2019

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677

(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of October 2018 is a recent version of Firefox from Mozilla.

11 Software Testing Concepts

11.1 Chapter Introduction

The goal of this chapter is to survey the important concepts, terminology, and techniques of software testing in general.

The next chapter illustrates these techniques by manually constructing test scripts for Haskell functions and modules.

11.2 Software Requirements Specification

The purpose of a software development project is to meet particular needs and expectations of the project's stakeholders.

By *stakeholder*, we mean any person or organization with some interest in the project's outcome. Stakeholders include entities that:

- have a "business" problem needing a solution—the project's sponsors, customers, and users
- care about the broad impacts of the project and its solution—that laws, regulations, standards, best practices, codes of conduct, etc., are followed
- are responsible for the development, deployment, operation, support, and maintenance of the software

A project's stakeholders should create a software requirements specification to state the particular needs and expectations to be addressed.

A *software requirements specification* seeks to comprehensively describe the intended behaviors and environment of the software to be developed. It should address the "what" but not the "how". For example, the software requirements specification should describe the desired mapping from inputs to outputs but not unnecessarily restrict the software architecture, algorithms, data structures, etc., that can be used to implement the mapping.

Once the requirements are sufficiently understood, the project's developers then design and implement the software system: its software architecture, its subsystems, its modules, and its functions and procedures.

Software testing helps ensure that the software implementation satisfy the design and that the designs satisfy the stakeholder's requirements.

Of course, the requirements analysis, design, and implementation may be an incremental. Software testing can also play a role in identifying requirements and defining appropriate designs and implementations.

11.3 What is Software Testing?

According to the Collins English Dictionary [Collins 2018]:

> A *test* is a deliberate action or experiment to find out how well something works.

The purpose of testing a program is to determine "how well" the program "works"—to what extent the program satisfies its software requirements specification.

Software testing is a "deliberate" process. The tests must be chosen effectively and conducted systematically. In most cases, the *test plan* should be documented carefully so that the tests can be repeated precisely. The results of the tests should be examined rigorously.

In general, the tests should be automated. Testers can use manually written test scripts (as we do in the next chapter) or appropriate testing frameworks (e.g. HUnit [HUnit 2018], QuickCheck [QuickCheck 2018], or Tasty [Tasty 2018] in Haskell).

Testers try to uncover as many defects as possible, but it is impossible to identify and correct all defects by testing. Testing is just one aspect of software quality assurance.

11.4 Goals of Testing

Meszaros [Meszaros 2007, Ch. 3] identifies several goals of test automation. These apply more generally to all software testing. Tests should:

- help improve software quality
- help software developers understand the system being tested
- reduce risk
- be easy to develop
- be easy to conduct repeatedly
- be easy to maintain as the system being tested continues to evolve

11.5 Dimensions of Testing

We can organize software testing along three dimensions [STF 2018]:

- testing levels
- testing methods
- testing types

We explore these in the following subsections.

11.5.1 Testing levels

Software *testing levels* categorize tests by the applicable stages of software development.

Note: The use of the term "stages" does not mean that this approach is only applicable to the traditional *waterfall* software development process. These stages describe general analysis and design activities that must be carried out however the process is organized and documented.

Ammann and Offutt [Ammann 2017] identify five levels of testing, as shown in Figure 11-1. Each level assumes that the relevant aspects of the level below have been completed successfully.

From the highest to the lowest, the testing levels are as follows.

1. *Acceptance testing* focuses on testing a completed system to determine whether it satisfies the software requirements specification and to assess whether the system is acceptable for delivery.

   The acceptance test team must include individuals who are strongly familiar with the *business requirements* of the stakeholders.

2. *System testing* focuses on testing an integrated system to determine whether it satisfies its overall specification (i.e. the requirements as reflected in the chosen software architecture).

   The system test team is usually separate from the development team.

3. *Integration testing* focuses on testing each subsystem to determine whether its constituent modules communicate as required. For example, do the modules have consistent interfaces (e.g. compatible assumptions and contracts)?

   A subsystem is often constructed by using existing libraries, adapting previously existing modules, and combining these with a few new modules. It is easy to miss subtle incompatibilities among the modules. Integration testing seeks to find any incompatibilities among the various modules.

   Integration testing is often conducted by the development team.

4. *Module testing* focuses on the structure and behavior of each module separately from the other modules with which it communicates.

   A module is usually composed of several *related units* and their associated data types and data structures. Module testing assesses whether the units and other features interact as required and assess whether the module satisfies its specification (e.g. its preconditions, postconditions, and invariants).

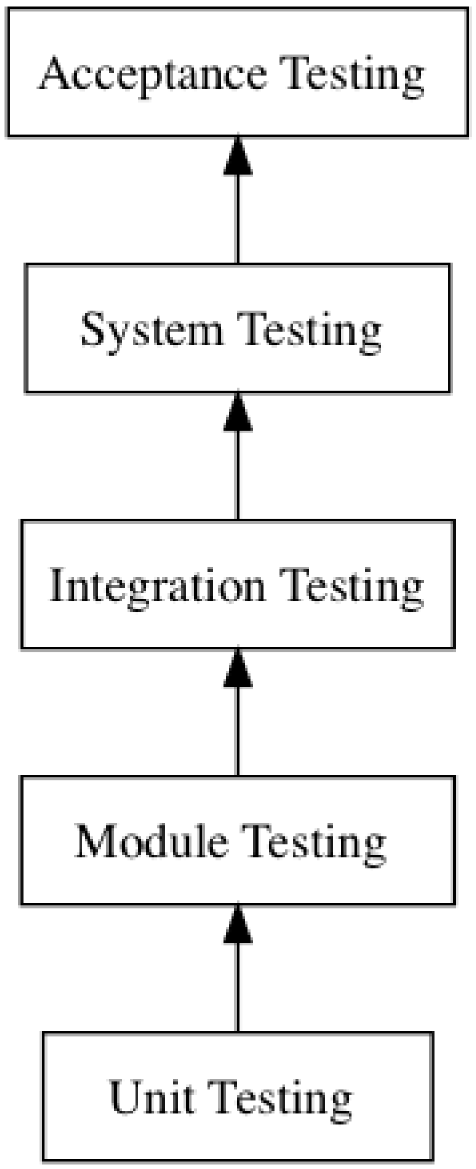

Figure 11-1. Software Testing Levels

Note: Here we use the term "module" generically. For example, a module in Java might be a `class`, `package`, or `module` (in Java 9) construct. A module in Python 3 might be a code file (i.e. module) or a directory structure of code files (i.e. package). In Haskell, a generic module might be represented as a closely related group of Haskell `module` files.

Module testing is typically done by the developer(s) of a module.

5. *Unit testing* focuses on testing the implementation of each program unit to determine whether it performs according to the unit's specification.

   The term "unit" typically refers to a procedural abstraction such as a function, procedure, subroutine, or method.

   A unit's specification is its "contract", whether represented in terms of preconditions and postconditions or more informally.

   Unit testing is typically the responsibility of the developer(s) of the unit.

In object-based systems, the units (e.g. methods) and the modules (e.g. objects or classes) are often tightly coupled. In this and similar situations, developers often combine unit testing and module testing into one stage called *unit testing* [Ammann 1997] [SFT 2018,

In this textbook, we are primarily concerned with the levels usually conducted by the developers: unit, module, and integration testing.

11.5.2 Testing methods

Software *testing methods* categorize tests by how they are conducted. The Software Testing Fundamentals website [STF 2018] identifies several methods for testing. Here we consider four:

- black-box testing
- white-box testing
- gray-box testing
- ad hoc testing

In this textbook, we are primarily concerned with black-box and gray-box testing. Our tests are guided by the contracts and other specifications for the unit, module, or subsystem being tested.

11.5.2.1 Black-box testing

In *black-box testing*, the tester knows the external requirements specification (e.g. the contract) for the item being tested but does not know the item's internal structure, design, or implementation.

Note: This method is sometimes called closed-box or behavioral testing.

This method approaches the system much as a user does, as a black box whose internal details are hidden from view. Using only the requirements specification and public features of the item, the testers devise tests that check input values to make sure the system yields the expected result for each. They use the item's regular interface to carry out the tests.

Black-box tests are applicable to testing at the higher levels—integration, systems, and acceptance testing—and for use by external test teams.

The method is also useful for unit and module testing, particularly when we wish to test the item against an abstract interface with an explicit specification (e.g. a contract stated as preconditions, postconditions, and invariants).

How do we design black-box tests? Let's consider the possible inputs to the item.

11.5.2.1.1 Input domain

An item being tested has some number of input parameters—some explicit, others implicit. Each parameter has some domain of possible values.

The *input domain* of the item thus consists of the Cartesian product of the individual domains of its parameters. A *test input* is thus a tuple of values, one possible value for each parameter [Ammann 2017].

For example, consider testing a public instance method in a Java class. The method has zero or more explicit parameters, one implicit parameter (giving the method access to all the associated instance's variables), and perhaps direct access to variables outside its associated instance (static variables, other instances' variables, public variables in other classes, etc.).

In most practical situations, it is impossible to check all possible test inputs. Thus, testers need to choose a relatively small, finite set of input values to test. But how?

11.5.2.1.2 Choosing test inputs

In choosing test inputs, the testers can fruitfully apply the following techniques [Everett 2007], [Meszaros 2007], [Perry 2006].

- Define *equivalence classes* (or partitions) of the possible inputs based on the kinds of behaviors of interest and then choose *representative* members of each class.

  After studying the requirements specification for the item being tested, the tester first groups together inputs that result in the "same" behaviors of interest and then chooses typical representatives of each group for tests (e.g. from the middle of the group).

The representative values are *normal use* or "happy path" cases that are not usually problematic to implement [Ammann 2017].

For example, consider the valid integer values for the day of a month (on the Gregorian calendar as used in the USA). It may be useful to consider the months falling into three equivalence classes: 31-day months, 30-day months, and February.

- Choose *boundary values*—values just inside and just outside the edges of an equivalence class (as defined above) or special values that require unusual handling.

  Unlike the "happy path" tests, the boundary values often are values that cause problems [Ammann 2017].

  For example, consider the size of a data structure being constructed. The boundary values of interest may be zero, one, minimum allowed, maximum allowed, or just beyond the minimum or maximum.

  For a mathematical function, a boundary value may also be at or near a value for which the function is undefined or might result in a nonterminating computation.

- Choose input values that *cover the range of expected results.*

  This technique works from the output back toward the input to help ensure that important paths through the item are handled properly.

  For example, consider transactions on a bank account. The action might be a balance inquiry, which returns information but does not change the balance in the account. The action might be a deposit, which results in a credit to the account. The action might be a withdrawal, which either results in a debit or triggers an insufficient funds action. Tests should cover all four cases.

- Choose input values *based on the model* used to specify the item (e.g. state machine, mathematical properties, invariants) to make sure the item implements the model appropriately.

  For example, a data abstraction should establish and preserve the invariants of the abstraction (as shown in the Rational arithmetic case study in Chapter 7).

Black-box testers often must give attention to tricky practical issues such as appropriate *error handling* and *data-type conversions.*

11.5.2.2 White-box testing

In *white-box testing*, the tester knows the internal structure, design, and implementation of the item being tested as well as the external requirements specification.

Note: This method is sometimes called open-box, clear-box transparent-box, glass box, code-based, or structural testing.

This method seeks to test every path through the code to make sure every input yields the expected result. It may use code analysis tools [Ammann 2017] to define the tests or special instrumentation of the item (e.g. a testing interface) to carry out the tests.

White-box testing is most applicable to unit and module testing (e.g. for use by the developers of the unit), but it can also be used for integration and system testing.

11.5.2.3 Gray-box testing

In *gray-box testing*, the tester has *partial* knowledge of the internal structure, design, and requirements of the item being tested as well as te external requirements specification.

Note: "Gray" is the typical American English spelling. International or British English spells the word "grey".

Gray-box testing combines aspects of black-box and white-box testing. As in white-box testing, the tester can use knowlege of the internal details (e.g. algorithms, data structures, or programming language features) of the item being tested to design the test cases. But, as in black-box testing, the tester conducts the tests through the item's regular interface.

This method is primarily used for integration testing, but it can be used for the other levels as well.

11.5.2.4 Ad hoc testing

In *ad hoc testing*, the tester does not plan the details of the tests in advance as is typically done for the other methods. The testing is done informally and randomly, improvised according the creativity and experience of the tester. The tester strives to "break" the system, perhaps in unexpected ways.

This method is primarily used at the acceptance test level. It may be carried out by someone from outside the software development organization on behalf of the client of a software project.

11.5.3 Testing types

Software *testing types* categorize tests by the purposes for which they are conducted. The Software Testing Fundamentals website [STF 2018] identifies several types of testing:

- *Smoke testing* seeks to ensure that the primary functions work. It uses of a non-exhaustive set of tests to "smoke out" any major problems.
- *Functional testing* seeks to ensure that the system satisfies all its functional requirements. (That is, does a given input yield the correct result?)
- *Usability testing* seeks to ensure that the system is easily usable from the perspective of an end-user.
- *Security testing* seeks to ensure that the system's data and resources are protected from possible intruders by revealing any vulnerabilities in the system
- *Performance testing* seeks to ensure that the system meets its performance requirements under certain loads.
- *Regression testing* seeks to ensure that software changes (bug fixes or enhancements) do not break other functionality.
- *Compliance testing* seeks to ensure the system complies to required internal or external standards.

In this textbook, we are primarily interested in functional testing.

11.5.4 Combining levels, methods, and types

A tester can conduct some type of testing during some stage of software development using some method. For example,

- a test team might conduct *functional testing* (a type) at the *system testing* level using the *black-box testing* method to determine whether the system performs correctly
- a programmer might do *smoke testing* (a type) of the code at the *module testing* level using the *white-box testing* method to find and resolve major shortcomings before proceeding with more complete functional testing

As noted above, in this textbook we are primarily interested in applying *functional testing* (type) techniques at the *unit, module, or integration testing* levels using *black-box or gray-box testing* methods. We are also interested in automating our tests.

11.6 Aside: Test-Driven Development

The traditional software development process follows a *design-code-test* cycle. The developers create a design to satisfy the requirements, then implement the design as code in a programming language, and then test the code.

Test-driven development (TDD) reverses the traditional cycle; it follows a *test-code-design* cycle instead. It uses a test case to drive the writing of code that satisfies the test. The new code drives the restructuring (i.e. refactoring) of the code base to evolve a good design. The goal is for the design and code to grow organically from the tests [Beck 2003] [Koskela 2013].

Beck describes the following “algorithm” for TDD [Beck 2003].

1. *Add a test* for a small, unmet requirement.

   If there are no unmet requirements, stop. The program is complete.

2. *Run all the tests.*

   If no tests fail, go to step 1.

3. *Write code to make a failing test succeed.*

4. *Run all the tests.*

   If any test fails, go to step 3.

5. *Refactor the code* to create a “clean” design.

6. *Run all the tests.*

   If any test fails, go to step 3.

7. Go to step 1 to start a new cycle.

Refactoring [Fowler 1999] (step 5) is critical for evolving good designs and good code. It involves removing duplication, moving code to provide a more logical structure, splitting apart existing abstractions (e.g. functions, modules, and data types), creating appropriate new procedural and data abstractions, generalizing constants to variables or functions, and other code transformations.

TDD focuses on functional-type unit and module testing using black-box and gray-box methods. The tests are defined and conducted by the developers, so the tests may not cover the full functional requirements needed at the higher levels. The tests often favor “happy path” tests over possible error cases [Ammann 2017].

This book presents programming language concepts using mostly small programs consisting of a few functions and modules. The book does not use TDD techniques directly, but it promotes similar rigor in analyzing requirements. As we have seen in previous chapters, this book focuses on design using contracts (i.e. preconditions, postconditions, and invariants), information-hiding modules, pure functions, and other features we study in later chapters.

As illustrated in the next chapter, these methods are also compatible with functional-type unit and module testing using black-box and gray-box methods.

11.7 Principles for Test Automation

Based on earlier work on the Test Automation Manifesto [Meszaros 2003], Meszaros proposes several principles for test automation [Meszaros 2007, Ch. 5]. These focus primarily on unit and module testing. The principles include the following:

1. *Write the tests first.*

   This principle suggests that developers should use Test-Driven Development (TDD) [Beck 2003].

2. *Design for testability.*

   Developers should consider how to test an item while the item is being designed and implemented. This is natural when TDD is being used, but, even if TDD is not used, testability should be an important consideration during design and implementation. If code cannot be tested reliably, it is usually bad code.

   The application of this principle requires judicious use of the abstraction techniques, such as those illustrated in Chapters 6 and 7 and in later chapters.

3. *Use the front door first.*

   Testing should be done primarily through the standard public interface of the item being tested. A test involves invoking a standard operation and then verifying that the operation has the desired result. (In terms of the dimensions of testing described in a previous section, this principle implies use of black-box and gray-box methods.)

   Special testing interfaces and operations may sometimes be necessary, but they can introduce new problems. They make the item being tested more complex and costly to maintain. They promote unintentional (or perhaps intentional) overspecification of the item. This can limit future changes to the item—or at least it makes future changes more difficult.

   Note: *Overspecification* means imposing requirements on the software that are not explicitly needed to meet the users' actual requirements. For example, a particular order may be imposed on a sequence of data or activities when an arbitrary order may be sufficient to meet the actual requirements.

4. *Communicate intent.*

   As with any other program, a test program should be designed, implemented, tested, and documented carefully.

   However, test code is often more difficult to understand than other code because the reader must understand both the test program and the item being tested. The "big picture" meaning is often obscured by the mass of details.

Testers should ensure they communicate the intent of a set of tests. They should use a naming scheme that reveals the intent and include appropriate comments. They should use standard utility procedures from the testing framework or develop their own utilities to abstract out common activities and data.

5. *Don't modify the system under test.*

   Testers should avoid modifying a "completed" item to enable testing. This can break existing functionality and introduce new flaws. Also, if the tests are not conducted on the item to be deployed, then the results of the tests may be inaccurate or misleading.

   As noted in principles above, it is better to "design for testability" from the beginning so that tests can be conducted through "the front door" if possible.

6. *Keep tests independent.*

   A test should be designed and implemented to be run independently of all other tests of that unit or module. It should be possible to execute a set of tests in any order, perhaps even concurrently, and get the same results for all tests.

   Thus each automated test should set up its needed precondition state, run the test, determine whether the test succeeds or fails, and ensure no artifacts created by the test affect any of the other tests.

   If one item depends upon the correct operation of a second item, then it may be useful to test the second item fully before testing the first. This dependency should be documented or enforced by the testing program.

7. *Isolate the system under test.*

   Almost all programs depend on some programming language, its standard runtime library, and basic features of the underlying operating system. Most modern software depends on much more: the libraries, frameworks, database systems, hardware devices, and other software that it uses.

   As much as possible, developers and testers should isolate the system being tested from other items not being tested at that time. They should document the versions and configurations of all other items that the system under test depends on. They should ensure the testing environment controls (or at least records) what versions and configurations are used.

   As much as practical, the software developers should encapsulate critical external dependencies within information-hiding components. This approach helps the developers to provide stable behavior over time. If necessary, this also enables the testers to substitute a “test double” for a problematic system.

A *test double* is a “test-specific equivalent” [Meszaros 2007, Ch. 11, 23] that is substituted for some component upon which the system under test depends. It may replace a component that is necessary but which is not available for safe use in the testing environment. For example, testers might be testing on system that interacts with another that has not yet been developed.

8. *Minimize test overlap.*

   Tests need to cover as much functionality of a system as possible, but it may be counterproductive to test the same functionality more than once. If the code for that functionality is defective, it likely will cause all the overlapping tests to fail. Following up on the duplicate failures takes time and effort that can better be invested in other testing work.

9. *Minimize untestable code.*

   Some components cannot be tested fully using an automated test program. For example, code in graphical user interfaces (GUIs), in multithreaded programs, or in test programs themselves are embedded in contexts that may not support being called by other programs.

   However, developers can design the system so that as much as possible is moved to separate components that can be tested in an automated fashion.

   For example, a GUI can perhaps be designed as a “thin” interactive layer that sets up calls to an application programming interface (API) to carry out most of the work. In addition to being easier to test, such an API may enable other kinds of interfaces in addition to the GUI.

10. *Keep test logic out of production code.*

    As suggested above, developers should “design for testability” through “the front door”. Code should be tested in a configuration that is as close as possible to the production configuration.

    Developers and testers should avoid inserting special *test hooks* into the code (e.g. `if testing then doSomething`) that will not be active in the production code. In addition to distorting the tests, such hooks can introduce functional or security flaws into the production code and make the program larger, slower, and more difficult to understand.

11. *Verify one condition per test.*

    If one test covers multiple conditions, it may be nontrivial to determine the specific condition causing a failure. This is likely not a problem with a manual test carried out by a human; it may be an efficient use of time to do fewer, broader tests.

However, tests covering multiple conditions are an unnecessary complication for inexpensive automated tests. Each automated test should be "independent" of others, do its own setup, and focus on a single likely cause of a failure. "

12. *Test concerns separately.*

    The behavior of a large system consists of many different, "small" behaviors. Sometimes a component of the system may implement several of the "small" behaviors. Instead of focusing a test on broad concerns of the entire system, testers should focus a test on a narrow concern. The failure of such a test can help pinpoint where the problem is.

    The key here is to "pull apart" the overall behaviors of the system to identify "small" behaviors that can be tested independently.

13. *Ensure commensurate effort and responsibility.*

    Developing test code that follows all of these principles can exceed the time it took to develop the system under test. Such an imbalance is bad. Testing should take approximately the same time as design and implementation.

    The developers may need to devote more time and effort to "designing for testability" so that testing becomes less burdensome.

    The testers may need to better use existing tools and frameworks to avoid too much special testing code. The testers should consider carefully which tests can provide useful information and which do not. There is no need for a test if it does not help reduce risk.

11.8 What Next?

This chapter surveyed software testing concepts.The next chapter applies them to testing Haskell modules from Chapters 4 and 7.

11.9 Exercises

TODO

11.10 Acknowledgements

I wrote this chapter in Summer 2018 for the 2018 version of the textbook *Exploring Languages with Interpreters and Functional Programming.*

- The discussion of the dimensions of software testing — levels, methods, and types — draws on the discussion on the Software Testing Fundamentals

website [STF 2018] and other sources [Ammann 2017] [Black 2007], [Everett 2007], [Perry 2006].

- The presentation of the goals and principles of test automation draws on the ideas of Meszaros et al [Meszaros 2003] [Meszaros 2007].
- The description of Test-Driven Development (TDD) "algorithm" is adapted from that of Beck [Beck 2003] and Koskela [Koskela 2013].

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

11.11 References

[**Ammann 2017**]**:** Paul Ammann and Jeff Offutt. *Introduction to Software Testing.* Cambridge University Press, 2017.

[**Beck 2003**]**:** Kent Beck. *Test-Driven Development: By Example*, Addison Wesley, 2003.

[**Black 2007**] Rex Black. *Pragmatic Software Testing*, Wiley, 2007.

[**Collins 2018**]**:** Collins Learning. Collins English Dictionary, https://www.collinsdictionary.com/us/dictionary/english/, accessed 28 June 2018.

[**Everett 2007**]**:** Gerard D. Everett and Raymond McLeod Jr. *Software Testing: Testing Across the Entire Software Development Life Cycle*, IEEE Press, 2007.

[**Fowler 1999**]**:** Martin Fowler. *Refactoring: Improving the Design of Existing Code*, Addison Wesley Professional, 1999.

[**HUnit 2018**]**:** Haskell Organization Hackage. HUnit: A Unit-Testing Framework for Haskell. http://hackage.haskell.org/package/HUnit, accessed 28 June 2018.

[**Koskela 2013**]**:** Lasse Koskela. *Effective Unit Testing*, Manning, 2103.

[**Meszaros 2003**]**:** Gerard Meszaros, Shaun M. Smith, and Jennitta Andrea. The Test Automation Manifesto, *Proceedings of the Conference on Extreme Programming and Agile Methods*, Springer, Berlin, 2003. Also at https://pdfs.semanticscholar.org/b42f/337557b91e5a4daa571fe19a8e937d9ac03d.pdf.

[**Meszaros 2007**]**:** Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*, Pearson Education, 2007.

[**Perry 2006**]**:** William E. Perry. *Effective Methods for Software Testing*, Third Edition, Wiley, 2006.

[**QuickCheck 2018**]**:** Haskell Organization Hackage. QuickCheck: Automatic Checking of Haskell Programs, http://hackage.haskell.org/package/QuickCheck, accessed 28 June 2018.

[**STF 2018**]**:** Software Testing Fundamentals (Dot Com), Software Testing Guide and Tutorial, http://softwaretestingfundamentals.com/, accessed 28 June 2018.

[Tasty 2018]: Haskell Organization Hackage. Tasty: Modern and Extensible Testing Framework, https://hackage.haskell.org/package/tasty, accessed 28 June 2018.

11.12 Terms and Concepts

Stakeholder, software requirements specification, test, test plan, testing dimensions (levels, methods, types), testing levels (unit, module, integration, system, and acceptance testing), testing methods (black-box, white-box, gray-box, and ad hoc testing), input domain, test input, input domain equivalence classes, representatives of normal use or "happy path", boundary values, covering the range of expected outputs, testing based on the specification model, error handling, data-type conversions, testing types (smoke testing, functional testing, usability testing, security testing, performance testing, regression testing, compliance testing), test-driven development (TDD), design-code-test vs. test-code-design.