Chapter 8

Software Testing
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Program testing

- The term “testing” can refer to two distinct activities:
  1. Attempting to demonstrate (often to the customer) that a program does what it is intended to do.
  2. Attempting to discover program defects – i.e., demonstrate that a program does NOT do what it is intended to do.

- Testing can reveal the presence of errors but (normally) NOT their absence.

- (Machine-based) testing is part of a more general verification and validation process, which also includes static V&V techniques (inspections, proofs of correctness, etc.).
Program testing goals

1. When the goal of testing is to demonstrate that software meets its requirements...
   - For custom software, this means that there should be at least one test for every user requirement in the SRS, plus combinations of these requirements.
   - For generic software, this means that there should be tests for all of the system features in the product release, plus combinations of these features.

(cont’d)
2. When the goal of testing is to discover situations in which the behavior of the software is incorrect, undesirable, or does NOT conform to its specification...
   • Tests should be designed to creatively root-out design and/or programming errors that could result in system crashes, unwanted interactions with other systems, poor performance, incorrect computations, data corruption, etc.

(cont’d)
Program testing goals (cont’d)

- Goal (1) is pursued via **validation testing**.
  - You expect the system to perform correctly and attempt to *demonstrate* this by using a given set of test cases that reflect the system’s expected use.
  - A “**successful test**” shows that the system operates as intended.

- Goal (2) is pursued via **defect testing**.
  - The test cases are *designed to expose defects*. Test cases used in defect testing can be deliberately obscure and need not reflect how the system is normally used.
  - A “**successful test**” reveals a program defect.
An input-output model of program testing
“Verification” vs. “Validation”

- **Verification:**
  - “Are we building the **product** right?”
  - The software should conform to its specification.

- **Validation:**
  - “Are we building the **right product**?”
  - The software should do what the user really needs / requires.
V&V confidence

- Verification and validation should establish confidence that the software is “fit for purpose.”
- This does NOT usually mean that the software must be completely free of defects.
- The level of confidence required depends on at least three factors...
Factors affecting level of confidence required

- **Software function / purpose:** Safety-critical systems, for example, require a much higher level of confidence than demonstration-of-concept prototypes.
- **User expectations:** Users may have low expectations or may tolerate shortcomings when the benefits of use are high.
- **Market environment:** Getting a product to market early may be more important than finding additional defects.
Ken Johnston’s “Minimum Viable Quality (MVQ)” testing model†

- Builds on premise that some companies test their web-based software services too much before releasing them to production:

  “You need to be comfortable with testing less and knowingly shipping buggier software faster than ever before. Speed of release is the vital competitive advantage in the world of connected services and devices.”

† Ken Johnston is a former Director of Test Excellence at Microsoft Corp.
MVQ testing model (cont’d)

Minimum Viable Quality (MVQ) Overview

Possible Testing

Increasing Test Investment

- Waste, excessive automation and excessive testing that does not find any meaningful bugs.
- Rich instrumentation identifies remaining critical to fix bugs in the shipped code.
- MVQ for all users but still use a rolling release process. Fix final few critical bugs after release
- MVQ for sub-set of users. Beta Users, Enthusiasts, Flighting
- Under Tested – frequent rollbacks, limited user engagement, strong negative customer feedback, bad press

Minimum Viable Quality

- Limited Release
- Release Quality
- MVQ for all users but still use a rolling release process. Fix final few critical bugs after release
- MVQ for sub-set of users. Beta Users, Enthusiasts, Flighting
- Under Tested – frequent rollbacks, limited user engagement, strong negative customer feedback, bad press

Under Tested
MVQ testing model (cont’d)

- “...new code is pushed to a subset of users; if it’s too buggy, a quick fail back to last known good takes the code out of use with minimum negative user impact.”
- “…you start to get data about how the code is functioning in production with real users more quickly.”
- “The key aspect is to balance...the feature set and (their) quality... If it is too low, then you won’t discover and learn the harder to find bugs because the code won't be exercised.”
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Inspections and testing

- **Software inspections / reviews:** analyzing static system representations such as requirements, design, source code, etc., to discover problems (static V&V) a.k.a. human-based testing
- **Software testing:** executing an implementation of the software to examine outputs and operational behaviour (dynamic V&V) a.k.a. machine-based testing

(cont’d)
Inspections and testing (cont’d)

Static V&V

Requirements specification

High-level design

Formal specification

Detailed design

Program

Prototype

Dynamic V&V

Program testing
Software inspections

- Involve people examining a system artifact (requirements, design documents, source code, etc.), usually with the aim of discovering anomalies and defects.
- *Can be more effective than testing* (after system implementation), as demonstrated in many studies.
Advantages of inspections

- Inspections *do not require execution* so they may be used *before system implementation*.
- During testing, one defect may *mask* others. (The behavior of the program *when encountering a defect* may obscure or completely eliminate any manifestation of other defects that would otherwise be observable.)
- With inspections, you needn’t be concerned with such *run-time interactions* among errors. (cont’d)
Advantages of inspections (cont’d)

- Incomplete (e.g., sub-) programs can be *inspected* without the specialized test harnesses required to *test* them.

- In addition to searching for program defects, an inspection team can also consider *some* broader quality attributes such as compliance with standards, portability, maintainability, etc.
Inspections and testing are complementary

- **Both** should be used during the V&V process.
- **Inspections** can be used early with non-executable entities and with source code at the module and component levels.
- **Testing** can validate dynamic behaviour and is the only effective technique at the sub-system and system (code) levels.
- **Inspections** cannot directly reveal some non-functional requirements issues such as those related to performance, usability, etc.
A model of the software (machine-based) testing process

“compare actual to expected results”
Stages of testing

- **Development testing**: the system is tested during development to discover bugs and defects.
- **Release testing**: a separate testing team tests a complete version of the system before it is released to users.
- **User testing**: users or potential users of a system test the system at the developer’s site (“alpha testing”) or in their own environment (“beta testing”).
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Development testing

- *Includes all testing activities that are carried out by the team developing the system…*

- **Unit testing**, where individual program units or object classes are tested independently.

- **Component testing**, where several individual units have been integrated to create (composite) components which are tested independently.

- **System testing**, where some or all of the components in a system have been integrated and tested as a whole.
Unit testing

- Unit testing is the process of testing individual program elements \textit{in isolation}.
- It is a \textit{defect testing} process.
- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods (more common)
Object class testing

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states (to the extent practicable).

- Inheritance makes it more difficult to design object class tests as the information to be tested is not localized.
The weather station object interface
Weather station testing

- Need to define test cases for `reportWeather()`, `reportStatus()`, etc.
- Using a state model, identify sequences of state transitions to be tested and the event sequences to cause these transitions.
- For example:
  - Shutdown → Running → Shutdown
  - Configuring → Running → Testing → Transmitting → Running
  - Running → Collecting → Running → Summarizing → Transmitting → Running
Automated testing

- Whenever possible, unit testing should be automated so that tests can be run and checked without manual intervention.
- A test automation framework (such as JUnit) can be used to write and run program tests.
- Such frameworks provide generic test classes that you extend to create specific test cases. They can run all of the implemented tests and report on the results.
Components of an automated test

- **A setup part**, whereby the system is initialized with the test case inputs and expected outputs.
- **A call part**, whereby the object or method to be tested is called.
- **An assertion part**, whereby the result of the call is compared to the expected result. If the assertion evaluates to true, no error has been detected.
Two types of tests

- Those that reflect the *normal behaviors specified for a program* in response to “valid inputs”.
- Those that reflect the *exceptional behaviors specified for a program* in response to “invalid inputs.” (E.g., outputting appropriate error messages, etc., instead of simply “crashing.”)
Testing Techniques

- **Black-Box**: Testing based solely on analysis of requirements (unit/component specification, user documentation, etc.). Also known as 
  *functional testing*.

- **White-Box**: Testing based on analysis of internal logic (design, code, etc.). (But *expected* results still come from requirements.) Also known as 
  *structural testing*. 
Testing strategies

- **Partition testing**: choose tests from identified groups of inputs that have common attributes and are therefore expected to be processed in the same way.

- **Guideline-based testing**: use testing guidelines reflecting previous experience with the kinds of errors that programmers often make to choose test cases.
Partition testing

- Input data often fall into different “equivalence partitions” where, based on the specification, all members of a partition would be expected to be processed in the same manner.
- Test cases should be chosen from each partition.
- A good rule of thumb is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partitions.
Partition testing (cont’d)
Partition testing example

Program Specification:

An ordered pair of numbers, \((x, y)\), are input and a message is output stating whether they are in **ascending order**, **descending order**, or **equal**. If the input is other than an ordered pair of numbers, an **error message** is output.

(cont’d)
Partition testing example (cont’d)

- Four “Equivalence Partitions”:
  \[
  \{ (x, y) \mid x<y \} \\
  \{ (x, y) \mid x>y \} \\
  \{ (x, y) \mid x=y \} \\
  \{ \text{input is other than an ordered pair of numbers} \} 
  \]
Guideline-based testing

Examples of general testing guidelines:

• Choose inputs that force the system to generate all error messages.
• Design inputs that cause buffers to overflow.
• Repeat the same input or series of inputs numerous times.
• Force invalid outputs to be generated.
• Force computation results to be too large (overflow) or too small (underflow).

(cont’d)
Guideline-based testing (cont’d)

- Guidelines for testing programs with lists or arrays:
  - Test with lists which have only a single value.
  - Use lists of different sizes in different tests.
  - Derive tests so that the first, middle, and last elements of the list are accessed.
  - Test with empty lists.
Component testing

- After individual objects have been unit tested in isolation, they are combined (integrated) to form “composite components”.
- Component-level testing then focuses on the behavior of the component as a single entity, as defined by the component interface specification.
Interface errors

- Potential inconsistencies in – or invalid assumptions about – the interfaces among newly integrated units are a principal focus of component testing. ("unit/object integration testing" would normally precede "component testing")

- The types of interfaces of interest include:
  - **Parameter interfaces**: the data passed from one method or procedure to another.
  - **Shared memory interfaces**: a block of memory is shared between procedures or functions.
  - **Procedural interfaces**: a sub-system encapsulates a set of procedures to be called by other sub-systems.
  - **Message passing interfaces**: sub-systems request services from other sub-systems.
Classes of interface errors (Lutz)

- **Interface misuse**: A calling component calls another component and makes an error in the use of its interface (e.g., parameters in the wrong order). *(syntactic)*

- **Interface misunderstanding**: A calling component embeds assumptions about the behaviour of the called component which are incorrect. *(semantic)*

- **Timing errors**: The called and the calling component operate at different speeds and out-of-date information is accessed.
Examples of general guidelines for interface testing

- Design tests so that parameters to a called procedure are at the extreme ends of their value ranges ("boundary value analysis").
- Always test pointer parameters with null pointers.
- Design tests which should cause the component to fail. (Applies to all levels of testing.)
- Use “stress testing” in message passing systems.
- In shared memory systems, vary the order in which components are activated.
System testing

- Involves integrating components to create a version of the system and then testing the integrated system as a whole.
  - May include reusable components that have been separately developed and/or off-the-shelf systems that have been integrated with newly developed components.
- A focus is testing the interactions between the (previously independently tested) components.
  - E.g., checking that components are compatible, interact correctly, and transfer the right data at the right time across their interfaces.

(cont’d)
System testing (cont’d)

- Also focuses on the functional and non-functional emergent behaviors of the system.
- In some development organizations, system testing may involve a separate testing team with no involvement from designers and programmers.
- The use-cases developed to identify system interactions can be used as a basis for system testing.
Test coverage policies

- "Exhaustive testing" is generally impossible, so testing policies which specify the required system test coverage may be developed. Examples include:
  
  - All system functions that are accessed through menus should be tested.
  
  - **Combinations of functions** (e.g., text formatting) that are accessed through the same menu must be tested.
  
  - Functions requiring user input must be tested with both correct ("valid") and incorrect ("invalid") input.
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Test-driven development (TDD)

- TDD was introduced in support of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

- Code is developed incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed (along with all other tests that have been implemented) passes its test.
TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement it as an automated test.
- Run the test, along with all other tests that have been implemented. (Initially, you have not implemented the functionality so the new test will “fail.”) Reinforces “test-first” mentality.
- Implement the functionality and re-run the test(s)...
- Once all tests run with no errors revealed, you move on to identifying the next increment of functionality.
TDD process model
Benefits of test-driven development

- **Code coverage**: Every increment of functionality written has at least one associated test case.

- **Regression testing**: A regression test suite is developed incrementally as a program is developed.

- **Simplified debugging**: When a test reveals an error, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

- **System documentation**: The tests themselves are a form of documentation that describe what the code should be doing.
Another perspective on the benefits of TDD...

“When you test first, you capture your intent in an automatable and executable form. You focus on what you are about to write in a way that works to prevent defects rather than create them. The tests you write serve as a persistent reinforcement of that intent going forward. In addition to helping you do the thing right, it helps you to do the right thing.”

— Stephen Vance, in Quality Code, 2014
Regression testing

- Regression testing is the RE-RUNNING of one or more test cases, after some program change, that ran without revealing faults prior to the change.
- It is used to determine if the change has “broken code” that worked correctly BEFORE the change.
- This can be very expensive in a manual testing environment, but with automation, it may be possible to run all previous (together with new) tests with each increment.
- TDD requires that tests must run without revealing errors before the change is committed.
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
Release testing...

- ...is testing a particular release of a system that is intended for use outside of the development environment.

- A separate team that has NOT been involved in system development should be responsible for release testing. Perhaps from a “QA” organization.

- Release testing is usually a black-box (i.e., functional) testing process.
Release testing vs. system testing

- While *system testing* focuses on discovering bugs in the system (=defect testing), the objective of *release testing* is to check that the system is good enough for external use (=validation testing).

- Therefore, release testing is intended to convince the supplier of the system that it will deliver its specified functionality, performance, and dependability, and that it will not fail during normal use.
“Requirements-based testing”

- A systematic form of release testing whereby a set of test cases is designed for each requirement.
- For example, consider a requirement for the patient info systems described in Chap. 1:
  - *If a patient is known to be allergic to any particular medication, then a prescription of that medication shall result in a warning message being issued to the system user.*
  - *If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.*
Associated requirement tests

- Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.

- Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to, and check that the warning is issued by the system.

- Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs and check that the correct warning for each drug is issued.

- Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.
“Scenario testing”

- A form of release testing whereby typical scenarios of use are used to develop test cases for the system.
- Consider, for example, a scenario describing one way that the patient info system described in Chap. 1 could be used on a home visit…
A usage scenario for the patient info system

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side-effect of keeping him awake at night. Kate looks up Jim’s record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim’s record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim’s record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.
Features tested from scenario

- Authentication by logging on to the system.
- Downloading and uploading of specified patient records to a laptop.
- Home visit scheduling.
- Encryption and decryption of patient records on a mobile device.
- Record retrieval and modification.
- Links with the drugs database that maintains side-effect information.
- The system for call list generation.
Performance testing

- Release testing may involve testing emergent properties of a system, such as performance.
- Tests should reflect the usage (“operational”) profile of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.
Topics covered

- Validation vs. defect testing
- Inspections and testing
- Development testing
  - Unit testing
  - Component testing
  - System testing
- Test-driven development
- Release testing
  - Requirements-based testing
  - Scenario testing
  - Performance testing
- User testing
User testing

- A stage in the testing process in which **users** or **customers** become directly involved in testing a system.

- **User testing is essential**, even when comprehensive system and release testing have been carried out.
  - Influences from the users’ working environment have a major effect on the reliability, performance, usability, and robustness of a system. These are difficult for developers to replicate.
Types of user testing

- **Alpha testing**: Users of the software work with the development team to test the software at the developer’s site.

- **Beta testing**: A release of the software is made available to users in the customer environment for experimentation and feedback.

- **Acceptance testing**: Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. *Primarily for custom systems.*
Stages in the acceptance testing process (user activities)

1. Define acceptance criteria
2. Plan acceptance testing
3. Derive acceptance tests
4. Run acceptance tests
5. Negotiate test results
6. Reject/accept system
Agile methods and acceptance testing

- With agile methods, there is no separate acceptance testing process.
- The user/customer defines tests that are integrated with other tests that are run automatically when changes are made.
- The main issue here is whether or not the embedded user is “typical” and can adequately represent the interests of other system stakeholders.
Key points

- Testing can only show the presence of errors in a program – not their absence. *(inductive process)*

- *Development testing* is the responsibility of the software development team, and includes *unit testing*, in which you test individual objects and methods, *component testing*, in which you test related groups of objects, and *system testing*, in which you test partial or complete systems.

(cont’d)
Key points (cont’d)

- In *defect testing*, you should try to “break” the software by using (among other techniques) experience and guidelines to choose types of test cases that have been effective in discovering defects in other systems.

- Whenever possible, *automated tests* embedded in a program that can be run every time a change is made to a system should be used.

- *Test-driven development* is an approach to development in which tests are designed, implemented, and *RUN(!) before* the code to be tested is designed and implemented.

(cont’d)
Key points (cont’d)

- **Scenario testing** involves inventing a typical usage scenario and using this to derive test cases.

- **Acceptance testing** is a user testing process where customers test a system to decide whether or not it is ready to be accepted and deployed in an operational environment.
Chapter 8
Software Testing