Software Testing

Chapter 8

Software Testing
in the textbook

- Introduction (Verification and Validation)
- 8.1 Development testing
- 8.2 Test-driven development
- 8.3 Release testing
- 8.4 User testing

Verification and Validation

- Verification:
  - The software should conform to its specification (functional and non-functional requirements).

    "Are we building the product right".
- Validation:
  - The software should do what the customer really requires.

    "Are we building the right product".

Requirements don't always reflect real wishes and needs of customers and users

Verification and Validation: Goals

- Establish confidence that the software is fit for purpose.
- NOT that it's completely free of defects.
  - generally not an achievable goal
- Good enough for its intended use
  - the type of use will determine the degree of confidence that is needed

Verification and Validation: confidence level

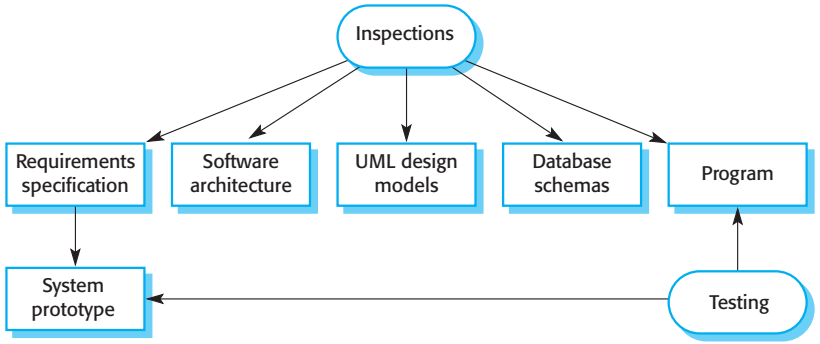
Required confidence level depends on:

- Software purpose
  - how critical is the software to an organization?
- User expectations
  - Users may have low expectations of certain kinds of software.
- Marketing environment
  - Getting a product to market early may be more important than finding all the defects in the program.

Verification Techniques

- Software inspections (static verification)
  - Analyze static system representation to discover problems
  - Specifications, design models, source code, test plans
- Software testing (dynamic verification)
  - The system is executed with simulated test data
  - Check results for errors, anomalies, data regarding non-functional requirements.

Verification Techniques

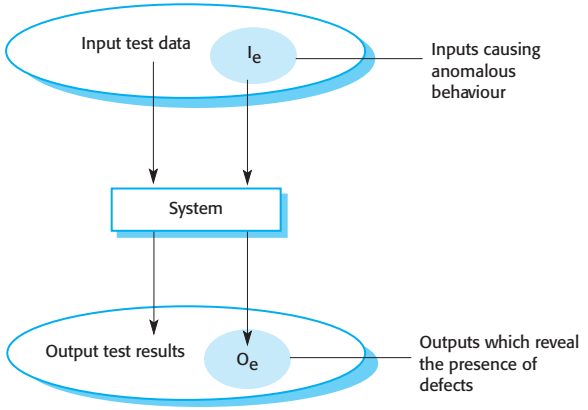


V&V at different stages in the software process

Software Testing Goals

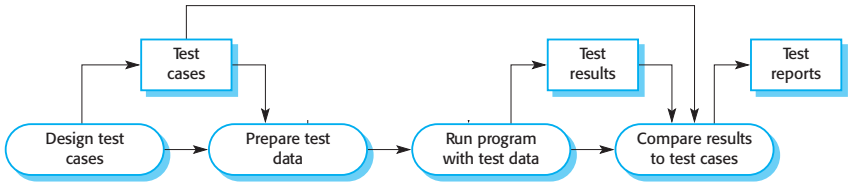
- Demonstrate that the software meets its requirements.
  - Validation testing
  - Collect test cases that show that the system operates as intended.
- Discover situations in which the behavior of the software is incorrect or undesirable.
  - Defect testing
  - Create test cases that make the system perform incorrectly.
  - Then fix the defect: Debugging

Software Testing



Validation testing (white only) versus Defect testing (blue)

Software testing process



Test Case specifies:
- what to test
- input data
- expected output

Can be automated

Three stages of software testing

- 8.1 Development testing:
  - the system is tested during development to discover bugs and defects.
- 8.3 Release testing:
  - a separate testing team tests a complete version of the system before it is released to users.
- 8.4 User testing:
  - users or potential users of a system test the system in their own environment.

8.1 Development testing

- All testing activities carried out by the team developing the system. (defect testing)
  - Unit testing: individual program units or object classes are tested.
    --should focus on testing the functionality of objects.
  - Component testing: several individual units are integrated to create composite components.
    --should focus on testing component interfaces.
  - System testing: some or all of the components in a system are integrated and the system is tested as a whole.
    --should focus on testing component interactions.

8.1.1 Unit testing

- Unit testing is the process of testing individual components in isolation.
  - functions vs classes
- Complete test coverage of a class:
  - Testing all operations associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states
- Inheritance makes it more difficult to design object class tests
  - The information to be tested is not localized.
  - Must test common info in each subclass.

Writing test cases for WeatherStation

1. Write a test case to check if identifier is set properly during normal system startup

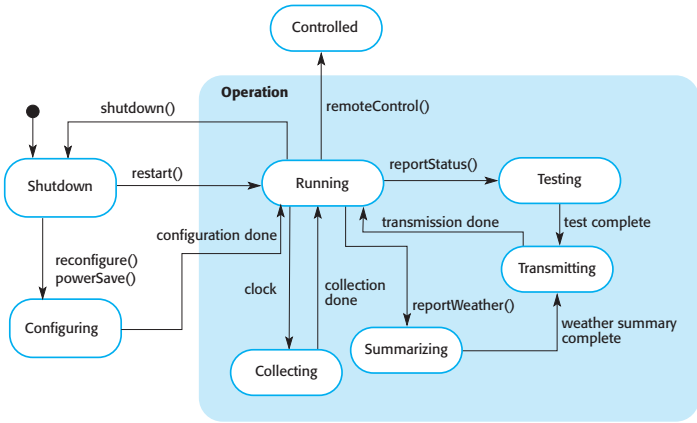
2. Write test cases for each of the methods (reportWeather(), etc.)
Try to test in isolation, but for example you cannot test shutdown unless you have already executed restart.

WeatherStation
identifier
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

Testing the states

- Use the state model:
  - Identify sequences of state transitions to be tested
  - Write a test case that generates the event sequences to cause these transitions.
  - Verify that the program ends up in the proper state.
- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

Testing States



Shutdown -> Running-> Shutdown is tested with a call to restart() followed by a call to shutdown(), then check the state

Automated testing

- Unit testing should be automated so that tests are run and checked without manual intervention.
  - JUnit or something like it
  - Your test classes extend the JUnit test class:
  - Initialize the test case with the **inputs** and **expected outputs**.
  - Call the method to be tested.
  - Make the assertion: compare the result of the call with the expected result.
  - When executed, if the assertion evaluates to true, the test has been successful if false, then it has failed.

Sample JUnit style test case (from ch. 3)

```
public class MoneyTest extends TestCase {

  public void testSimpleAdd() {
    Money m1 = new Money(12,"usd");
    Money m2 = new Money (14, "usd");
    Money expected = new Money(26, "usd");
    Money result = m1.add(m2);
    assertEquals (expected, result);
  }
}
```


8.1.2 Choosing unit test cases

Two types of unit test cases:

- show the component behaves correctly under normal use
  - For example, demonstrate (part of) a use case
- expose the defects/bugs
  - For example, invalid input should generate error message and fail gracefully

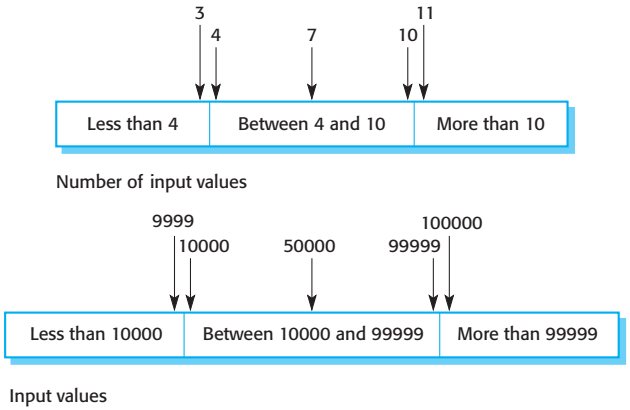
Strategies for choosing unit test cases

- **Partition testing**: identify groups of inputs that have common characteristics and should be processed in the same way.
  - choose tests from within each of these groups.
- **Guideline-based testing**: use testing guidelines based on the kinds of errors that programmers often make.
  - choose tests based on these guidelines.

Partition testing

- Divide the input data of a software unit into partitions of data
  - program should behave similarly for all data in a given partition
- Determine partitions from specifications
- Design test cases to cover each partition at least once.
- If the partition is a range, also test boundary values.
- Enables good test coverage with fewer test cases.

Equivalence partitions example



Specification states the program accepts 4 to 10 inputs which are five-digit integers (greater than 10,000)

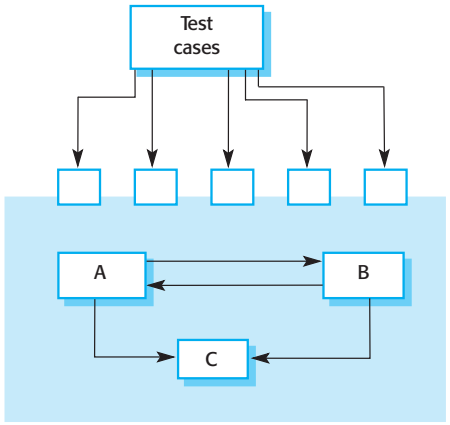
Guideline-based testing

- Choose test cases based on previous experience of common programming errors
- For example:
  - Choose inputs that force the system to generate all error messages
  - Repeat the same input or series of inputs numerous times
  - Try to force invalid outputs to be generated
  - Force computation results to be too large or too small.
  - Test sequences/lists using
    - one element
    - zero elements
    - different sizes in different tests

8.1.3 Component testing

- Software components are made up of several interacting objects (or sub-components)
- The functionality of these objects is accessed through the defined component interface.
- Component testing is demonstrating that the component interface behaves according to its specification.
  - Assuming the subcomponents (objects) have already been unit-tested

Interface (component) testing



Small empty boxes represent the interface

Common interface errors

- Interface misuse
  - Calling component uses another component's interface incorrectly e.g. parameters in the wrong order.
- Interface misunderstanding
  - Calling component embeds assumptions about the behavior of the called component which are incorrect.
  - eg passing unordered array to binary search
- These are usually errors in the CALL to the component, not within the component, so they are tested during system testing.

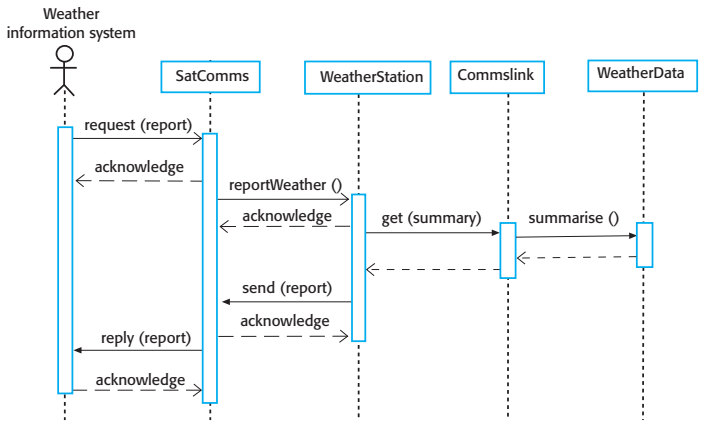
8.1.4 System testing

- System testing: integrating components to create a version of the system and then testing the integrated system.
- Checks that:
  - components are compatible,
  - interact correctly
  - transfer the right data at the right time across their interfaces.
- Tests the interactions between components.

Use-case testing

- Use-cases developed to identify system interactions are a good basis for system testing.
- The sequence diagrams documents the components and interactions that are being tested.

reportWeather sequence diagram



Shows which components are involved in a test case that calls SatComms:request(report)

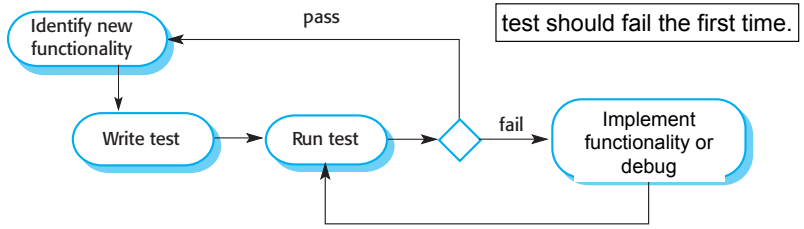
Testing policies

- Exhaustive system testing is impossible
- Testing policies define the required subset of all possible test cases, for an organization.
- Examples of testing policies:
  - 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 in various sequences.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

8.2 Test-driven development

- An approach to program development (not testing) in which you inter-leave testing and code development.
- Tests are written before code, in small increments.
- Don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced in agile methods but it can also be used in plan-driven development processes.

Test-driven development



Benefits of test-driven development

- Code coverage
  - All code written has at least one associated test.
- Regression testing
  - A regression test suite is developed incrementally as a program is developed.
  - Run these after each change, to make sure nothing got broken
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

8.3 Release testing

- Testing a particular release of a system that is intended for use outside of the development team.
- Primary goal: convince the supplier of the system that it is good enough for use.
- Similar to system testing, but
  - Tested by a team other than developers.
  - Focus is on regular use (not defects).
- Usually a black-box testing process where tests are only derived from the system specification.

8.3.1 Requirements-based testing

- Examine each requirement in the SRS and develop a test (or tests) for it.
- Example requirements from MHC-PMS system:
  - If a patient is known to be allergic to any particular medication, then 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.

Tests developed to test the requirement

- 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 to that the patient is allergic to, and check that the warning is issued by the system.
- Also write tests for more than one known allergy, prescribing more than one drug, etc.
- 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.

8.3.2 Scenario testing

- A scenario is a story that describes one way in which the system might be used
  - Longer than an “interaction”, may be several interactions
  - May have scenarios that were used during requirements engineering.
- To use a scenario for release testing:
  - tester assumes role of user, acting out scenario
  - may make deliberate mistakes
  - takes note of problems (slow response, etc.)
- Tests several requirements at once, in combination.

Scenario from MHC-PMS

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 of those patients who she has to contact for follow-up information and make clinic appointments.

Features tested by the 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 prompting.

8.3.3 Performance testing

- Designing and running tests to show the system can process its intended load.
- Use a typical operational profile: a set of tests that reflect the actual mix of work that will be handled by the system.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.
- The load is steadily increased until the system performance becomes unacceptable.
- Stress testing often required for distributed systems.

8.4 User testing

- Users or customers provide input and advice on system testing.
  - formal process where user tests a custom system or
  - informal process where user experiments with system
- User testing is essential, even when comprehensive system and release testing have been carried out.
  - Influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system.
  - These cannot be replicated in a testing environment.

Types of user testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
  - custom software
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to report problems
  - usually generic software
- Acceptance testing
  - Customers test a system to decide whether or not it is ready to be accepted from the system developers.
  - acceptance implies payment is due, may require negotiation.
  - custom software.