Software Testing

Massimo Felici

JCMB-1402 0131 650 5899

1BP-G04 0131 650 4408

mfelici@inf.ed.ac.uk

What is Software Testing?

Software Testing *is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs*

Terminology

- **Fault**: an imperfection that may lead to a failure
  - E.g., missing/incorrect code that may result in a failure
  - **Bug**: another name for a fault in code
- **Error**: where the system state is incorrect but may not have been observed
- **Failure**: some failure to deliver the expected service that is observable to the user

Testing Goals

- **Validation testing**
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended
- **Defect testing**
  - To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system

Effectiveness vs. Efficiency

- **Test Effectiveness**
  - Relative ability of testing strategy to find bugs in the software
- **Test Efficiency**
  - Relative cost of finding a bug in the software under test

What is a successful test?

- **Pass**
  - Status of a completed test case whose actual results are the same as the expected results
- **No Pass**
  - Status of a completed software test case whose actual results differ from the expected ones
  - "**Successful**" test (i.e., we want this to happen)

Software Testing Features

- The **scope** of testing
  - The different levels of the system that testing addresses
- **Test techniques**
  - Some of the approaches to building and applying tests
- **Test management**
  - How we manage the testing process to maximize the effectiveness and efficiency of the process for a given product

Testing scope

- **"Testing in the small"** (**unit testing**)
  - Exercising the smallest executable units of the system
- **"Testing the build"** (**integration testing**)
  - Finding problems in the interaction between components
- **"Testing in the large"** (**system testing**)
  - Putting the entire system to test

Testing Categorization

- **Fault-directed** testing
  - Unit testing
  - Integration testing
- **Conformance-directed** testing
  - System testing
  - Acceptance testing

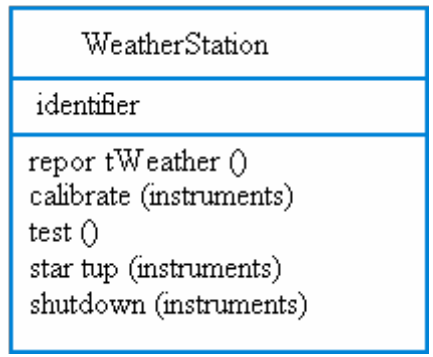
Testing "in the small"

- **Unit Testing**
  - Exercising the smallest individually executable code units
  - It is a defect testing process.
  - Component or unit testing is the process of testing individual components in isolation.
- Components may be
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality
- Objectives
  - Finding faults
  - Assure correct functional behaviour of units
- Usually performed by programmers

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
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

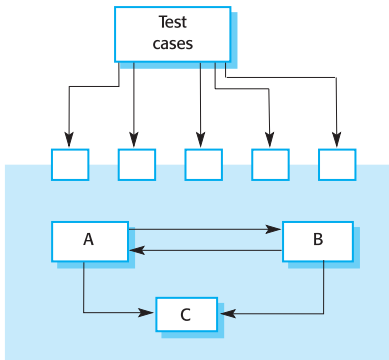
An Example of Object Class Testing



- Need to define test cases for
  - reportWeather, calibrate, test, startup and shutdown
- Using a state model, identify sequences of **state transitions** to be tested and the event sequences to cause these transitions
- For example
  - Waiting -> Calibrating -> Testing -> Transmitting -> Waiting

Interface Testing

- Objectives are to detect faults due to **interface errors** or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces



Interface Errors

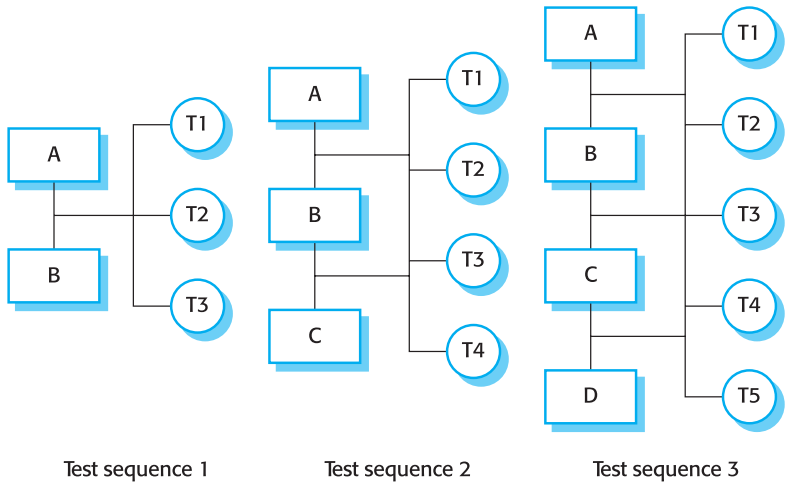
- **Interface misuse** - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order
- **Interface misunderstanding** - A calling component embeds assumptions about the behaviour of the called component which are incorrect
- **Timing errors** - The called and the calling component operate at different speeds and out-of-date information is accessed

Testing the "build"

- **Integration Testing**
  - Exercising two or more units or components
- **Objectives**
  - Detect interface errors
  - Assure the functionality of the combined units
- Performed by **programmers** or **testing group**
- **Issues**
  - Strategy for combining units?
  - Compatibility with third-party components (e.g., Commercial Of The Shelf - COTS)?
  - Correctness of third-party components?

Integration Testing

- Involves building a system from its components and testing it for problems that arise from **component interactions**.
- Top-down integration
  - Develop the skeleton of the system and populate it with components.
- Bottom-up integration
  - Integrate infrastructure components then add functional components.
- To simplify error localisation, systems should be **incrementally integrated**.



Testing "in the large": System

- **System Testing**
  - Exercising the functionality, performance, reliability, and security of the entire system
- **Objectives**
  - Find errors in the overall system behaviour
  - Establish confidence in system functionality
  - Validate non-functional system requirements
- Usually performed by a separate **testing group**

Testing "in the large": Accept

- **Acceptance Testing**
  - Operating the system in the user environment with standard user input scenario
- **Objectives**
  - Evaluate whether the system meets the customer criteria
  - Determine whether the customer will accept the system
- Usually performed by the **end user**

Testing "in the large": Operation

- **Regression Testing**
  - Testing modified versions of a previously validated system
- **Objectives**
  - Assuring that changes to the system have not introduced new errors
- Performed by the system itself or by a regression test group
- Capture/Replay (CR) Tools

Test Generation Methods

- **Black-box testing**
  - No knowledge of the software structure
  - Also called specification-based or functional testing
- **White-box testing**
  - Knowledge of the software structure and implementation
  - White-box methods can be used for test generation and test adequacy analysis
  - Usually used as adequacy criteria (after generation by a black-box method)
  - Methods based on internal code structure: **Statement**, **Branch**, **Path** or **Data-flow coverage**
- **Fault-based testing**
  - Objective is to find faults in the software, e.g., Unit testing
- **Model-based testing**
  - Use of a data or behaviour model of the software, e.g., finite state machine
- **Random testing**

Structural Testing

- **Statement Testing**: requires that very statements in the program be executed
- **Branch Testing**: seeks to ensure that every branch has been executed.
  - Branch coverage can be checked by probes inserted at the points in the program that represent arcs from branch points in the flowgraph.
  - This instrumentation suffices for statement coverage as well.
- **Expression Testing**: requires that every expression assume a variety of valued during a test in such a way that no expression can be replaced by a simpler expression and still pass the test.
  - Expression testing does require significant runtime support for the instrumentation.
- **Path Testing**: data is selected to ensure that all paths of the program have been executed.
  - In practice, path coverage is impossible to achieve

Issues with Structural Testing

- Is code **coverage** effective at detecting faults?
- How much coverage is enough?
- Is one coverage criterion better than another?
- Is coverage testing more effective that random test case selection?

Test Management

- **Concerns**
  - Attitude to testing
  - Effective documentation and control of the whole test process
  - Documentation of tests and control of the test codebase
  - Independence of test activities
  - Costing and estimation of test activities
  - Termination: deciding when to stop
  - Managing effective reuse
- **Activities**
  - Test Planning
  - Test case generation – can involve massive amounts of data for some systems
  - Test environment development
  - Execution of tests
  - Evaluating test results
  - Problem reporting
  - Defect tracking

From Use Cases to Test Cases

A (Black-box) Tester's Viewpoint

- What is the system supposed to do?
- What are the things that can go wrong?
- How can I create and record a set of testing scenarios?
- How will I know when to stop testing?
- Is there anything else the system is supposed to do?

From Use Cases to Test cases

- One of the greatest benefits of use cases is that they provide a set of assets that can be used to drive the testing process
- Use cases can directly drive, or seed, the development of test cases
- The scenarios of a use case create templates for individual test cases
- Adding data values completes the test cases
- Testing non-functional requirement completes the testing process

Deriving Test Cases from Use Cases

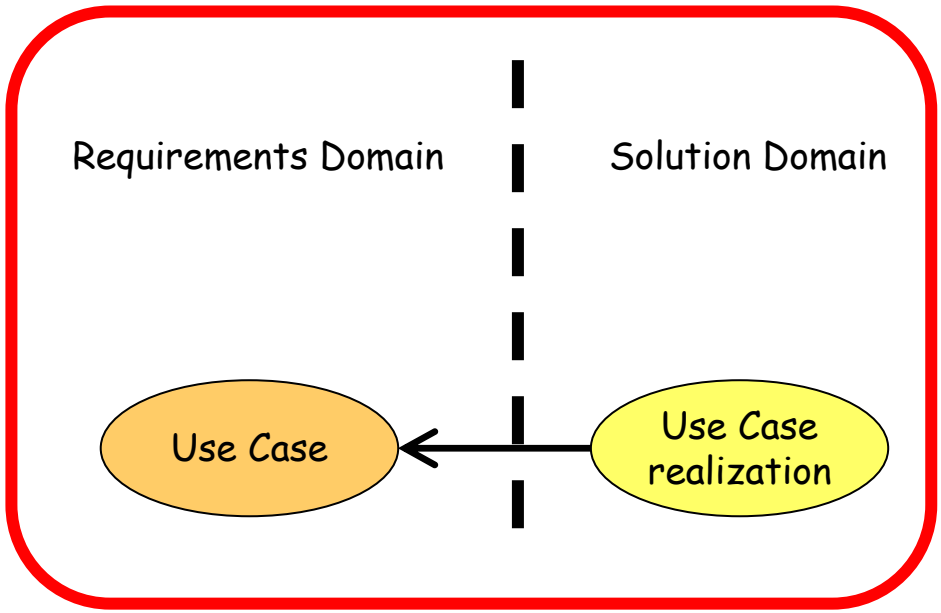
1. Identify the use-case scenarios
2. For each scenario, identify one or more test cases
3. For each test case, identify the conditions that will cause it to execute
4. Complete the test case by adding data values

Managing Test Coverage

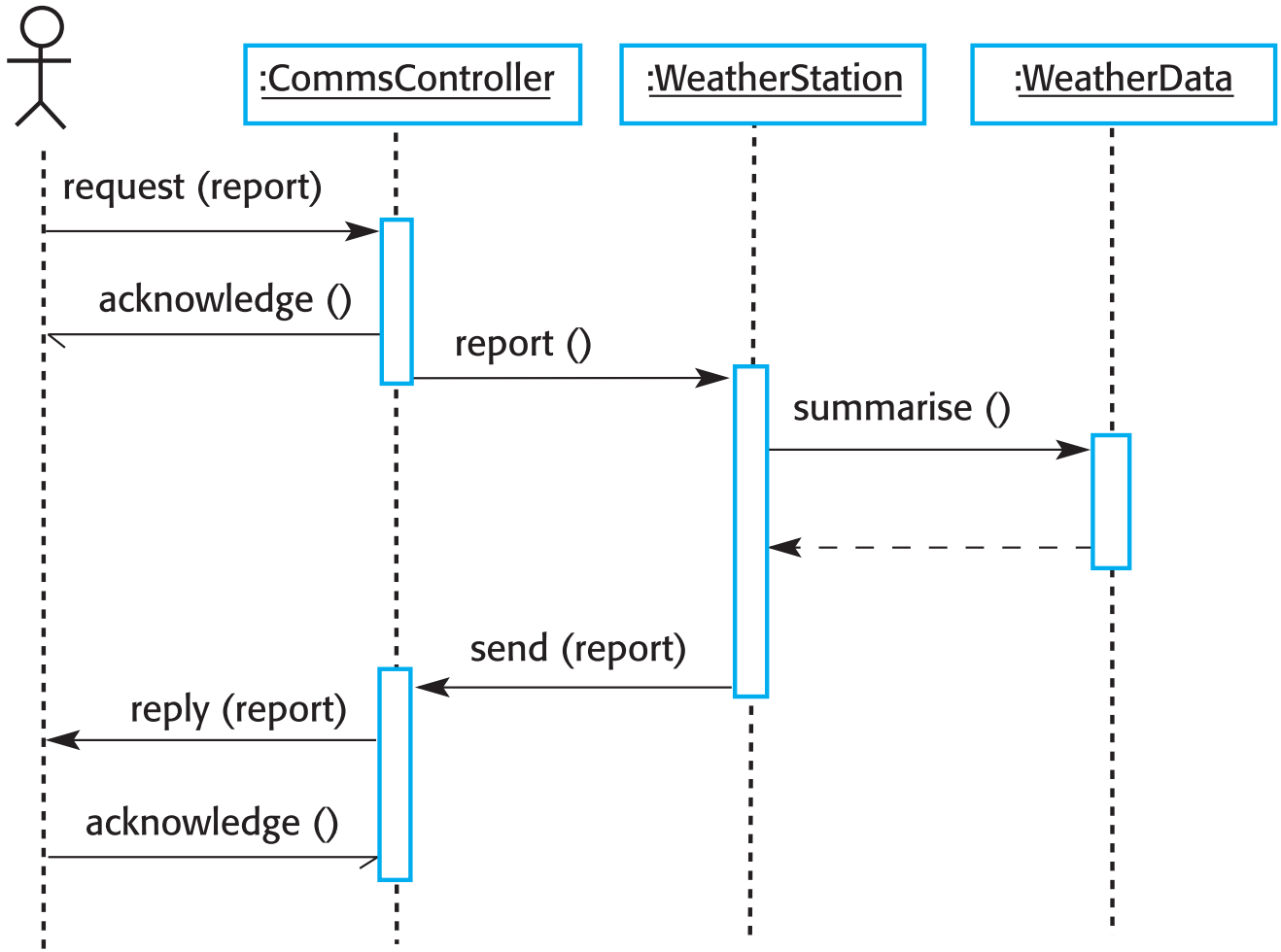
- Select the most appropriate or critical use cases for the most thorough testing
  - Often these use cases are primary user interfaces, are architecturally significant, or present a hazard or hardship of some kind to the user should a defect remain undiscovered
- Chose each use case to test based on a balance between cost, risk, and necessity of verifying the use case
- Determine the relative importance of your use cases by using priorities specific to your context

Black-box vs. White-box Testing

- For every use case, there is a use case realization that represents how the system is designed to accomplish the use case
- The use case itself lives in the requirements domain and simply specify necessary behaviour
- The use-case realization lives inside the solution space and describes how the behaviour is accomplished by the system



An Example of Use Case-based Testing



Is a Use Case a Test Case?

- NO
- Test cases
  - Test cases form the foundation on which to design and develop test procedures
  - The "depth" of the testing activity is proportional to the number of test cases
  - The scale of the test effort is proportional to the number of use cases
  - Test design and development, and the resources needed, are largely governed by the required test cases
- Use-case Scenarios
  - A scenario, or an instance of a use case, is a use-case execution wherein a specific user executes the use case in a specific way

A Matrix for Testing Specific Scenarios

Test Case ID	Scenario / Condition	Description	Data Value 1 / Condition 1	Data Value 2 / Condition 2	…	Expected Result	Actual Result
1	Scenario 1						
2	Scenario 2						
3	Scenario 2						

Readings

- James A. Whittaker. What is Software Testing? And Why is it so Hard?. In IEEE Software, January/February 2000, pp. 70-79.

Suggested Readings

- P.C. Jorgensen, C. Erickson. Object Oriented Integration Testing. Communications of the ACM, September 1994.

Summary

- Testing is a critical part of the development of any system
- Testing can be carried out at a number of levels and is planned as an integral part of the development process
- There is a wide range of approaches to test case generation and evolution of the adequacy of a test suite
- Test needs to be managed effectively if it is to be efficient