Section 6
Testing

1. Concepts
2. Unit testing
3. Integration testing
4. System testing

Testing Outcomes

- Learning outcomes
  - understand the main categories of testing
  - unit testing:
    - understand creation of test cases for blackbox and whitebox testing
  - integration testing:
    - select testing integration strategy
    - understand creation of test cases using stubs and drivers
  - system testing:
    - understand creation of test cases based on the functional model

Section 6.1
Testing Concepts

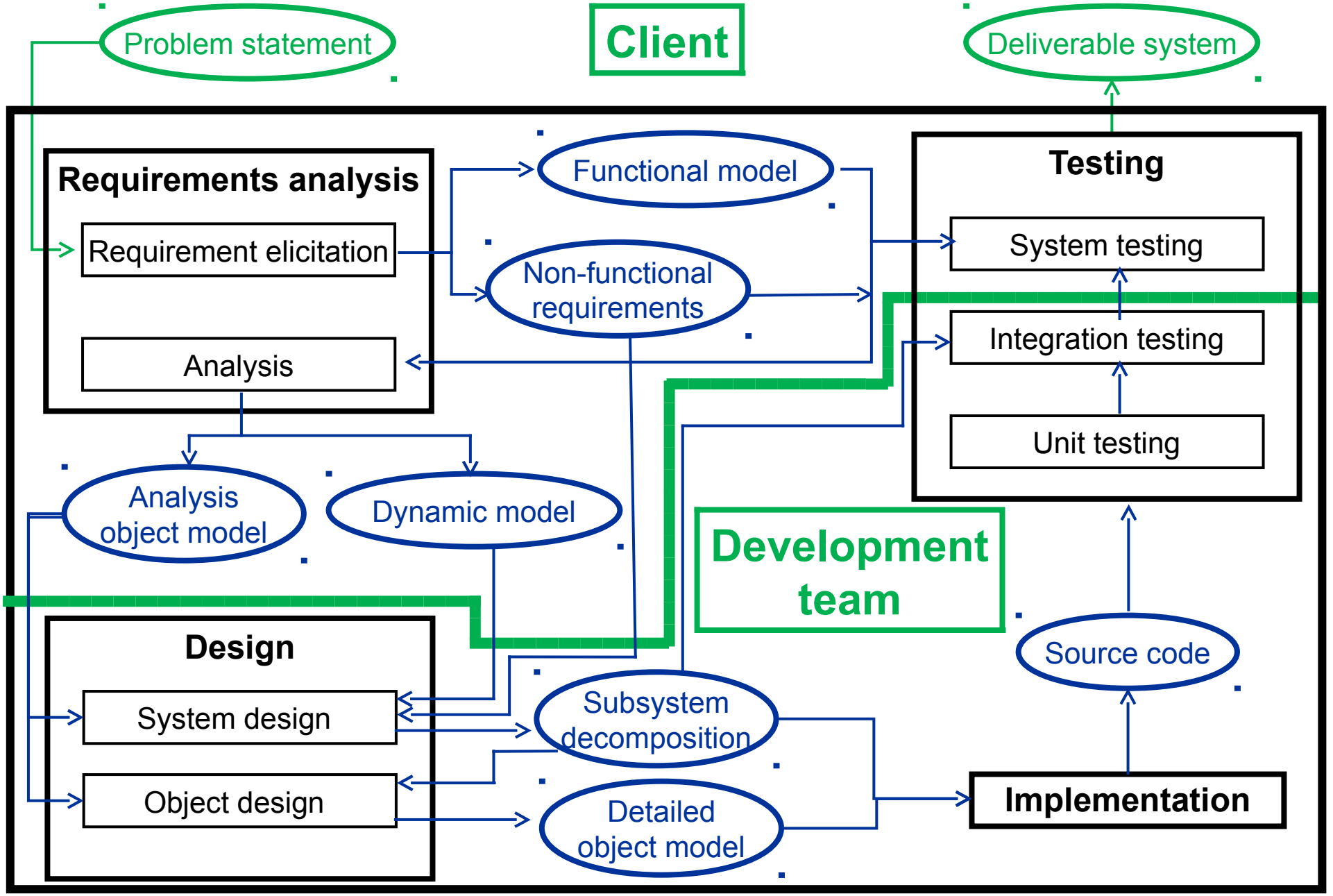
1. Overview
2. Basic concepts
3. Usability testing

6.1.1 Overview

- Input to testing
  - functional model
  - non-functional requirements
  - subsystem decomposition
  - source code

- Output from testing
  - deliverable system

Testing Work Products



Overview (cont.)

- What is testing?
  - it’s the process of finding differences between:
    - the *specified* system behaviour
    - the *observed* system behaviour
  - it’s the systematic attempt to find faults in a planned manner
- Purpose
  - to break the system
  - testing is based on the falsification of the system models

Overview (cont.)

- How do we test software?
  - we design tests that expose the defects in the system
- Who does the testing?
  - developers who are not involved in the design or implementation
  - specialized testers

Overview (cont.)

- Terminology
  - failure:
    - deviation of the observed behaviour from the specified behaviour
  - error state:
    - system is in a state where further processing will lead to failure
  - fault:
    - a defect or bug
    - the mechanical or algorithmic cause of an error

6.1.2 Basic Concepts

- Software reliability
- Code reviews
- Testing approach
- Blackbox and whitebox testing
- Faults, error states, failures
- Test cases
- Test stubs and drivers
- Corrections

Software Reliability

- What is reliability?
  - the degree to which observed behaviour conforms to specification
- Techniques for increasing reliability
  - fault avoidance
  - fault detection
  - fault tolerance

Software Reliability (cont.)

- Fault avoidance
  - detect faults statically, without model execution
  - prevent faults before system is released
  - includes:
    - use of development methodologies
    - configuration management
    - verification

Software Reliability (cont.)

- Fault detection
  - identify error states and faults before release
  - includes
    - debugging (uncontrolled)
    - testing (controlled)
- Fault tolerance
  - cope with faults and system failures
  - recover from faults and failures at runtime
    - for example, using redundant components

Code Reviews

- What is a code review?
  - manual inspection of parts of the system without execution
  - performed by a review team
- Two types of review
  - walkthrough
    - developer presents the code
  - inspection
    - review team checks everything

Code Reviews (cont.)

- Goal of code reviews
  - to check code against the requirements
    - both functional and non-functional
  - to check for algorithm efficiency
  - to check accuracy and completeness of comments

Code Reviews (cont.)

- Code inspections:
  - take place with other developers and quality assurance people
  - can uncover a lot of bugs
  - are time-consuming
  - crucial for safety-critical projects

- Inspections include:
  - preparation time, where participants study the code
  - formal meeting, where reviewers:
    - debate possible faults
    - look for inefficiencies
    - ensure code adheres to coding conventions and standards

Testing Approach

- How should we approach testing?
  - demonstrate that the system has faults
  - select test data that has a high probability of finding faults
- What if your testing doesn't find faults?
  - it's not that the code is good
  - it's that the tests are not thorough enough
- Main things to check
  - wide range of inputs
  - invalid inputs (e.g. outside range, zeros, wrong data type)
  - boundary cases

Testing Approach (cont.)

- Overall testing activities
  - test planning
    - test cases can be designed as soon as the models are stable
  - usability testing
    - test the UI design
  - unit testing
    - test the objects and subsystems of individual use cases
  - integration testing
    - test the individual components in combination
    - includes structural testing, which tests all components together

Testing Approach (cont.)

- Overall testing activities (cont.)
  - system testing
    - test the system as a whole
    - test all the scenarios, requirements, and design goals
  - system testing includes:
    - functional testing
      - based on Requirements Analysis Document
    - performance testing
      - based on System Design Document
    - acceptance testing
      - checks the system against the project agreement
      - performed by the client

Blackbox and Whitebox Testing

- Test component
  - part of the system that is isolated for testing
  - can include one object, group of objects, one or more subsystems

- Blackbox testing
  - test cases are **not** based on the code's internal structure
  - the focus is on input/output behaviour of the test component
  - ignores the internal aspects of the test component
    - both behaviour and structure

Blackbox and Whitebox Testing (cont.)

- Whitebox testing
  - test cases are based on the code's internal structure
  - the focus is on the internal structure of the test component
  - ensures that everything is tested:
    - every state in the dynamic model
    - all the object interactions

- Unit testing must include **both** blackbox and whitebox

Faults, Error States, Failures

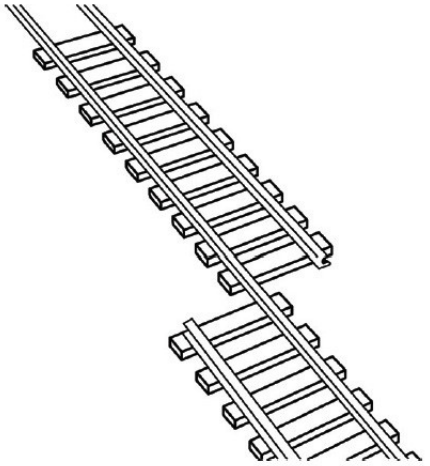


Figure 11-3 An example of a fault. The desired behavior is that the train remain on the tracks.

Use case name	`DriveTrain`
Participating actor	`TrainOperator`
Entry condition	`TrainOperator` pushes the "StartTrain" button at the control panel.
Flow of events	1. The train starts moving on track 1. 2. The train transitions to track 2.
Exit condition	The train is running on track 2.

Figure 11-4 Use case `DriveTrain` specifying the expected behavior of the train.

Faults, Error States, Failures (cont.)

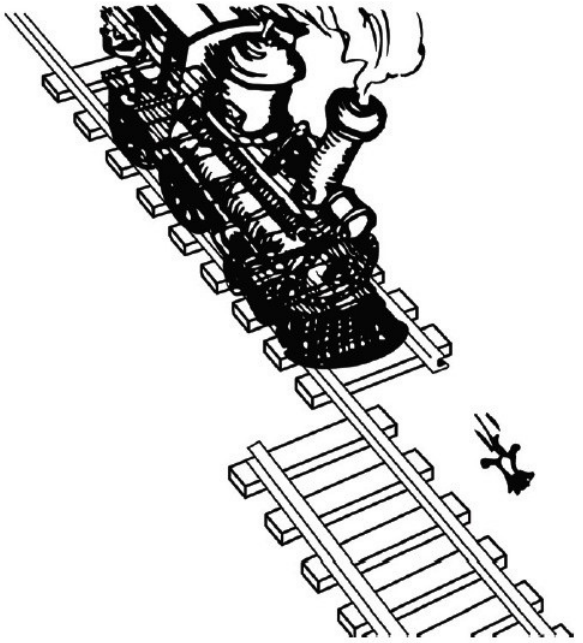


Figure 11-6 An example of an erroneous state.

Faults, Error States, Failures (cont.)

- Types of faults
  - algorithmic
    - may be introduced during implementation, design, or analysis
    - examples:
      - data structure overload
      - lack of initialization
      - performance problems
  - mechanical
    - occur even if the implementation is correct
    - examples:
      - fault in the programming environment
      - power failure
- Failure in one component can lead to failure in another

Test Cases

- What is a test case?
  - a set of inputs and expected results that exercise a component
  - the goal is to cause failures and detect faults

- Every test case has these attributes:
  - a unique name
    - it should be derived from the associated requirement or component
  - the component under test
    - the operations, classes, and/or subsystems being tested
  - input
    - set of input data or commands entered by actor (tester or test driver)
  - expected output
    - expected results against which the output of the test is compared

Test Cases (cont.)

- Test case development dependencies
  - functional testing
    - based on the functional model (use cases)
  - unit testing
    - based on the definition of subsystem interfaces
- Test cases can be developed as soon as models are stable
- Testing can be parallelized

Test Cases (cont.)

- Test cases can be related by associations
  - aggregation
    - a test case can be broken down into a set of sub-tests
  - precedence
    - used to specify when one test case must precede another
- Test case associations should be minimized
  - this allows for test cases to be run in parallel

Test Cases (cont.)

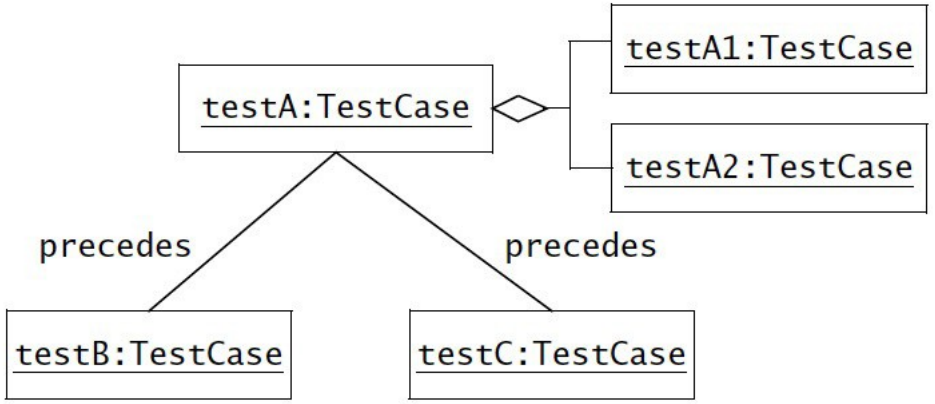


Figure 11-9 Test model with test cases. TestA consists of two tests, TestA1 and TestA2. TestB and TestC can be tested independently, but only after TestA has been performed.

Test Stubs and Drivers

- Issue:
  - we need to test the test component **in isolation** of the rest of the code
  - problem:
    - some piece of code has to *call the test component*
    - some code has to execute when the test component *calls other code*
  - we need to **substitute** the missing parts
    - we can't use the real code because it hasn't been tested yet

- Solution
  - we write **test drivers** that call the test component
  - we write **test stubs** that simulate the called portions of code

Test Stubs and Drivers (cont.)

- Test driver
  - simulates the part of the system that *calls* the test component
  - passes in the test inputs to the test component
  - displays the results of the test

- Test stub
  - simulates the component that *is called by* the test component
  - provides the same API as the simulated component
  - must return the expected value
  - must simulate the exact behaviour or may cause test case failure
  - may be more efficient to use than the actual called component

Corrections

- What is a correction?
  - a change to a component in order to repair a fault
- Danger:
  - introducing a new fault with the correction

Corrections (cont.)

- Managing corrections
  - problem tracking
    - documentation of errors and code fixes
  - regression testing
    - re-execution of all prior tests after a code fix
    - ensures that all previously working functionality is intact
    - this should be as automated as possible
  - rationale maintenance
    - documentation of reasons for change
    - ensures that no new faults are introduced by violating previous assumptions

6.1.3 Usability Testing

- Focus of usability testing
  - finding differences between the system and user expectations
- Possible problems
  - UI details
  - layout of screens
  - sequence of interactions
  - hardware
- Possible approaches
  - developers set out test objectives
  - participants accomplish predefined tasks
  - developers observe and collect user performance data

Usability Testing (cont.)

- Three types of usability tests
  - scenario test
    - users are presented with a scenario of the system
    - developers gauge user reactions to how scenario models work tasks
    - this can be achieved with story boards or with prototypes
  - prototype test
    - users presented with software that implements key aspects of system
    - vertical prototype: one use case is completely implemented
    - horizontal prototype: a single layer is implemented
      - for example: UI prototype
  - product test
    - users presented with a functional system