Testing

In this lecture

We will:

- See how to devise a black-box test plan

Testing

- Testing software is a very important activity because it provides information about the quality of the code being tested

- Testing can show whether a piece of code…
  - Satisfies the software requirements
  - Functions as expected

- Finding a fault earlier in development is cheaper to correct than discovering it later
  - Careful testing reveals faults
  - Testing should be done early

Types of testing

- There are different types of software testing including:
  - Functional
    - Does the code do what it should?
  - Non-functional
    - Does the code measure up to other kinds of requirements e.g. response times, usability, reliability
  - Black-box
    - With known input, do we get the expected output?
    - i.e. this does not consider how the code is implemented
  - White-box
    - Has every code statement been executed at least once?
    - i.e. this considers the internal data structures and algorithms

When is testing done?

- Black-box tests can be planned from the beginning of the project
  - i.e. when the software requirements have been written
- Black-box test plans can be used at any stage of the lifecycle
  - A design can be evaluated against the entire test plan
  - The application code can be evaluated against some or all of the test plan depending on the level of testing being conducted

Complete testing is impossible

- A software application can **never** be tested to exhaustion

- Consider a program that sums two integers
  - There is an infinite number of integers that could be input to the program
  - Therefore, there is an infinite number of tests that should be performed to prove that the program is correct
  - This is not possible to do in practice

Equivalence partitioning

- Equivalence partitioning divides the set of all possible input values into groups of equivalent values, and then uses at least one value to represent each group
  - It is assumed that if a test passes with the representative values from the group, then all values from that group would pass the test
- Consider:
  - Add two integers
    - One group: all integers
  - Add two positive integers
    - Two groups: all positive integers; all negative integers

Black-box testing

- For each functional requirement, use equivalence partitioning to devise **test cases** that cover each of the following:
  - **Normal values**
    - Values that are drawn from the set of valid values for the application
  - **Invalid values**
    - Values that are drawn from the set of invalid values, the application should detect these as invalid
  - **Boundary values**
    - Values that test the boundaries between partitions
  - **Special cases**
    - Values that have not been considered by the other tests e.g. remove last item from a stack

Prefix the case number with N (e.g. N12)

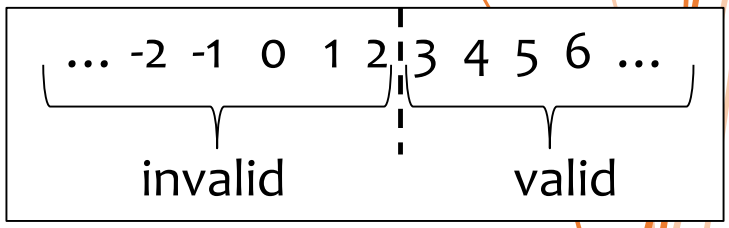
Prefix the case number with I (e.g. I12)

Prefix the case number with B (e.g. B12)

Prefix the case number with S (e.g. S12)

Boundary values test cases

- A boundary exists between two adjacent equivalence groups
  - Use two test cases for a boundary, one each side of the boundary
- For the requirement `num must be greater than two`, we can identify the following
  - There are two partitions
    - Invalid values: **two** and below
    - Valid values: **three** and above
  - The values either side of the boundary are **two** and **three**
  - There are two boundary test cases:
    - B1: `num = 2`
    - B2: `num = 3`



The test plan

- A test plan consists of one or more tests
- A test consists of:
  - A unique identifier
    - for ease of reference
  - A set of instructions
    - The sequence of operations to perform in the test
  - A list of the test cases addressed by the test
    - To show the reason for choosing the test data
  - A description of the expected outcome of the test
    - So the tester can compare actual results against the expected results

Test coverage

- A single test covers:
  - A single invalid test case

    or
  - A single special test case

    or
  - One or more normal or boundary test cases

- Invalid and special cases are tested in isolation to avoid the possibility that one error masks another

Best practice in testing

- Pick data values for which the expected outcome can be easily determined
  - e.g. it is easier to determine the expected output of

    ```
    7 * 2
    ```

    than

    ```
    35463 * 453
    ```

- Use different values for each input data item
  - e.g. when testing the calculation of the volume of a cuboid, input data

    ```
    length=2, width=2, height=2
    ```

    would give the expected outcome even if the calculation were incorrect

    e.g. `length * length * height`

Best practice in testing

- Do not assume an order of the tests in the plan
  - Write each test to be independent of all other tests
    - This avoids side effects
    - One test does not affect another, possibly masking problems
- Document tests
  - Fully describe each test showing the reason for it, and the expected outcome
- Keep test plan as small as possible while ensuring all cases are covered
  - This avoids unnecessary tests

Test plan example

Problem

A sports team has a name, and consists of a number of players. When the team has the required number of players, it can compete.

A player has a name, a team and a position. The position may be “Attack” or “Defence”, and is set only when the player is allocated to a team. If the player is not allocated to a team, the position must be blank (i.e. “”).

A player can only be in one team at a time. A player cannot be added to a team if he or she is already in the team.

A team cannot have more than the required number of players, although it can have fewer.

A team’s required number of players must be one or more.

Write and test Java classes to implement these requirements.

Developing a test plan

1. Write a list of requirements drawn from the problem description

2. For each requirement, identify, where appropriate, the following test cases:
    1. Normal
    2. Invalid
    3. Boundary
    4. Special

3. Write a test plan based on the test cases

Develop a test plan – partial example

Requirement

R1: A team has a name

R2: A team has a required number of players, greater than zero

Test cases

N1: Team name = “A-Team” (R1)

N2: Required number of players = 5 (R2)

I1: Team name = “” (R1)

I2: Required number of players = -4 (R2)

B1: Required number of players = 0 (R2)

B2: Required number of players = 1 (R2)

Develop a test plan – partial example

Test plan

UT1: Create a team called “A-Team” requiring 5 players
(N1, N2)
Expected: There is a team whose name is “A-Team” requiring 5 players

UT2: Create a team with a blank name requiring 5 players
(I1)
Expected: Error – Team not created

UT3: Create a team called “Z-Team” requiring -4 players
(I2)
Expected: Error – Team not created

Develop a test plan – partial example

Test plan (cont.)

UT4: Create a team called "Y-Team" requiring 0 players (B1)
Expected: Error – Team not created

UT5: Create a team called "B-Team" requiring 1 player (B2)
Expected: There is a team whose name is "B-Team" requiring 1 player

The full test plan (`17_Lec_TestPlan.docx`), which can be downloaded from Blackboard

Implementing the test plan

- One way of implementing a test plan is to write an application class
  - Each test is written in its own method, outputting the actual results
  - The `main()` method calls each test in turn

In the project `TestingExample`, there is an example of testing in the class `TestPlanImplementedByApplication`

- Advantage
  - Conceptually very easy

- Disadvantage
  - Success or failure of a test can only be detected by detailed visual inspection of the output
    - What if there are 10,000 tests in the plan?

You should be able to…

- Devise a black-box test plan