CISC327 - Software Quality Assurance

Lecture 10
Black Box Testing
Black Box Testing

• Outline
  – Last time we explored the role of systematic testing in the **software life cycle**
  – Today we continue with:
    • Introduction to testing methods: **black box and white box**
    • Kinds of **black box** methods
    • Black box method 1: Systematic **functionality testing**
Systematic Testing Methods

• For a test method to be systematic, it must have:
  – a system (rule) for creating tests
  – a measure of completeness

• Need an easy, systematic way to create test cases
  – to know for sure what to test

• Need an easy, systematic way to run tests
  – to know how to test

• Need an easy, systematic way to decide when we're done
  – to know when we have enough tests
Black Box vs. White Box

• Two Kinds of Methods
  – Systematic testing methods can be divided into two kinds: black box and white box
  – Black box methods cannot see the software code to test it (it may not exist yet), so they can only base their tests on the requirements or specifications
  – White box methods can see what the software's code is, so they can base their tests on the software's actual architecture or code itself
Black Box Methods

• Black box testing is for **functionality**
  – No knowledge of the internal structure or source code is necessary
Black Box Methods

• Black Box Methods
  – In a black box method, we choose our test cases based solely on the requirements, specification, or (sometimes) design documents
  – Advantage: we can do it independently of the software
    • On a large project, black box tests can be developed in parallel with the development of the software, saving time
  – Normally, black box testing is based on the functional specification (requirements) for the software system
Black Box Methods

• Functional Specifications
  – Can be formal (mathematical), or more often informal (in a natural language such as English)
  – In either case, the functional specification usually contains at least three kinds of information:
    • the intended inputs
    • the corresponding intended actions
    • the corresponding intended outputs
  – Focussing on each one of these separately gives us three different black box systems for testing
Black Box Methods

• Three Kinds of Black Box Methods
  – Systematic black box methods can be divided into three classes corresponding to these three kinds of information:
    • Input coverage tests
    • Output coverage tests
    • Functionality coverage tests
Black Box Methods

• **Input coverage** tests
  – Based on an analysis of the intended inputs, independent of their actions or outputs

• **Output coverage** tests
  – Based on an analysis of the intended outputs, independent of their input or actions

• **Functionality coverage** tests
  – Based on an analysis of the intended actions, with or without their inputs and outputs
Systematic Functionality Testing

• An example

  – We begin with the third of these, functionality coverage testing

  – Functionality coverage attempts to partition the functional specification of the software into a set of small, separate requirements

  – Example: Suppose that the informal requirements for a program we are to write are as follows:

    • "Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."
Systematic Functionality Testing

• **Requirements Partitioning**
  – Our first step is to physically **partition** the functional specification into separate requirements
  – In this system we **model** the separate requirements as independent, even though they are not
  – Example:
    • "Given as input two integers x and y, output all the numbers smaller than or equal to x that are evenly divisible by y. If either x or y is zero, then output zero."
Requirements Partitioning

• "Given as input two integers x and y"
  – R1. Accept two integers as input.
• "output ... the numbers"
  – R2. Output zero or more (integer) numbers.
• "smaller than or equal to x"
  – R3. All numbers output must be less than or equal to the first input number.
• "evenly divisible by y"
  – R4. All numbers output must be evenly divisible by the second number.
• "all the numbers"
  – R5. Output must contain all numbers that meet both R3 and R4.
• "If either x or y is zero, then output zero."
  – R6. Output must be zero (only) in the case where either first or second input integer is zero.
Test Case Selection

• **Test Cases for Each Requirement**
  
  – We model each partitioned requirement as **independent**
  
  – We create separate **test cases** for each partitioned requirement
  
  – **Example**: For the partitioned requirement:
    
    • "If either x or y is zero, then output zero."

    **R6.** Output must be zero (only) in the case where either first or second input integer is zero.
Test Case Selection

— **Example**: For the partitioned requirement:
  - "If either x or y is zero, then output zero."
  - **R6**: Output must be zero (only) in the case where either first or second input integer is zero.

— We might choose the test cases:
  - **R6T1**: 0 0 (both zero)
  - **R6T2**: 0 1 (x zero, y not)
  - **R6T3**: 1 0 (y zero, x not)
  - **R6T4**: 1 1 (neither zero)

— Notice these test inputs are the **simplest possible** and make no attempt to be exhaustive - more on this later
A Systematic Method

• **Black Box Functionality Coverage**
  – Functionality coverage gives us a *system* for creating functionality test cases
    • It tells us when we are *done* (i.e., when we have test cases for every partitioned requirement)
  – But notice this is **not** the same as *acceptance* testing, because it treats functional requirements as if they were completely *separate*, when in fact they are tightly *related*
    • So it does not replace acceptance testing, we (or the customer) must do that *as well*
    • Unlike acceptance testing, it is a *systematic* method, but like other systematic methods, it is only a *partial* test
Choosing & Organizing Tests

• An Experiment
  – Black box testing performs an experiment on the software system, in the true scientific sense
  – We have a hypothesis that the software has certain properties, and we design a method to test whether the hypothesis holds with our test cases
  – We then observe the results and draw conclusions, in classic scientific method style
Choosing & Organizing Tests

• Experimental Design
  – A fundamental principle of experimental design is the isolation of "variables"
    • This refers to the fact that we should design the experiment such that each possible cause that may affect the outcome (each experimental "variable") can be observed independently
    • Thus when an effect is observed, we can tell which cause is at work
  – The usual way to do this is to design the experiment in steps that only vary one "variable" (possible cause) at a time, keeping everything else constant
Choosing & Organizing Tests

• Test Plan Design
  – The experimental model of software testing gives us two important principles for our test plan:
    1. Test inputs should be chosen to carefully isolate different causes of failure (the experimental variables)
    2. Test cases should be ordered such that each test only assumes features to be working that have already been tested by a previous test
Guidelines for Choosing Test Inputs

• Choosing Inputs
  – For test inputs, this principle means that we should help isolate failure causes, by as much as possible
    1. Consistently choosing the simplest input values possible, in order not to introduce arbitrary variations
    2. Keeping everything constant between test cases, varying only one input value at a time
       (don't try to be "clever" introducing random input variations)
  – These principles hold for all systematic test methods, not just this one
Ordering Tests for QBASIC

• **T2** can log in in ATM mode

• **T14** createacct disallowed in ATM mode
Ordering Tests for QBASIC

• **T14** createacct disallowed in ATM mode
  
  .
  
  .
  
  .

• **T2** can log in in ATM mode

If T14 doesn’t produce the expected result, is it because *(a)* createacct is erroneously allowed, or *(b)* because ATM login doesn’t work?
Don’t Duplicate Tests

• Some parts of the project requirements are redundant:
  – “after an ATM login, only ATM transactions and logout are accepted”
  – [createacct] “privileged transaction, only accepted in agent mode”

• This is a natural way of stating the requirements, not an inconsistency

• But don’t write two identical test cases here
Check your levels

• We are not doing acceptance testing here
• Instead, breaking down the requirements into pieces and testing each one
• None of the test cases you’re writing for A1 will be big enough to seem “realistic”
Assumptions

• Front End behaviour is deterministic: same input ⇒ same output.
• Can you think of a kind of testing that isn’t deterministic?
• Aside: “voting” on airplanes
• Implementors (you!) are reasonable:
  – You won’t write code that detects the specific case of 27 createacct transactions in a row so you can crash on purpose
Summary

• Black Box Testing
  – Two classes of systematic test methods, black box and white box
  – Black box methods include input coverage, output coverage, functionality coverage
  – Functionality coverage partitions the functional specification into separate requirements to test
  – Isolate causes by ordering tests by features used, keeping test input values simple, and varying one input value at a time
Summary

• References
  – Sommerville, ch. 8, "Software Testing"

• Next Time
  – More Black Box Testing - Input coverage methods