

CISC 327 - Software Quality Assurance

Lecture 13

Black Box Unit Testing

Black Box Unit Testing

- Black box **method** testing
 - Test **harnesses**
 - Role of code-level specifications (**assertions**)
 - **Automating** black box unit testing
- Black box **class** testing ("**interface testing**")
 - Class **trace** specifications
 - **Executable** assertions

Black Box Unit Testing

- We can use black box testing to test individual **units** (methods or classes) as we develop them
- Because this kind of unit testing is **black box**, we still have the advantage that tests can be created in **parallel** with coding
- Moreover, black box **unit** testing is earlier and more precise than black box **system** testing
 - It can find errors very early, even before the entire first version is finished

Black Box Unit Testing

- To use black box testing on code units, we need to figure out what corresponds to **requirements**, **inputs**, and **outputs** for units
- Once we know these, test cases can be created according to any of the black box testing criteria:
 - **functionality** coverage
 - **input** coverage
 - **output** coverage

Black Box Testing of **Single Methods**

- If the unit we are testing is a **method** (or function or procedure), then:
 - “**requirements**” = the method’s specification
 - “**input**” = value of parameters, instance variables, global variables used by the method
 - “**output**” = return value, changed instance variables, global variables, and thrown exceptions; together these are referred to as the **outcome** in unit testing

Sidebar: “or function or procedure”

- In most **functional** languages, the **input** to a function consists of
 - the parameters (arguments) of the function
 - the values of any “free variables” mentioned in the function
- In contrast, object-oriented languages always have an implicit “zeroth argument” (**this** or **self**)

Sidebar: “or function or procedure”

- In **pure (= no side effects) functional** languages, the **outcome** of the unit test consists of
 - the return value
- With side effects, we have more outputs to worry about:
 - changed instance variables
 - changed global variables
 - thrown exceptions
 - file I/O, graphics, network connections, ...

Sidebar: “or function or procedure”

- We could argue about whether disallowing side effects is a good choice in general, but it does seem to make unit testing easier!
- Leads to tools like the QuickCheck test case generator (first released in 1999 for Haskell, since ported to other languages)

Black Box Testing of **Single Methods**

1. Identify the *input*
2. Identify the *output (= outcome)*

Black Box Testing of **Single Methods**

1. Identify the *input*

- method parameters
- instance variables
- global variables
- file I/O, standard input, network reads, ...

2. Identify the *output (= outcome)*

- changed instance variables
- changed global variables
- file I/O, standard output, network writes, ...
- return value **or** thrown exception

Black Box Testing of Single Methods

- For the **push()** method in a **Stack** class implementing stacks of integers
 - **void push (int x)**
throws (stackOverflow)
- The **input** is...
- The **output** is...

Black Box Testing of Single Methods

- For the **push()** method in a **Stack** class implementing stacks of integers
 - **void push (int x)**
throws (StackOverflow)
- The **input** is the current state of the **Stack** object and the parameter value of **x** to push onto it
- The **output** is the new state of the **Stack** object and the exception thrown
- We can then apply, for example, **input partitioning** to create test partitions:
 - P1 **Stack** empty, **x = 1**
 - P2 **Stack** nonempty, **x = 1**
 - P3 **Stack** full, **x = 1**

Executing Unit Tests

- Once we have test cases for a method, we need a way to run them
- A **method** is not a standalone **program**, so we need a special standalone program that calls the method: a **test harness**, which **automates** testing by:
 - Running test cases
 - Generating test reports

Executing Unit Tests

- The test harness sets up an appropriate **environment** to use the unit and invokes it with test input, checking the test **outcome**, and reporting any failure
- The set of test cases are **programmed** in the harness as a sequence of **calls** or uses of the tested unit with the test inputs, with code to check the outcome after each call

Test Harnesses

```
// A test harness to test the push() method
import Stack;
public class TestPush {
    public static void main (String[] args) {
        // Create a Stack object
        Stack s = new Stack();

        // P1 - empty Stack, X=1
        s.Push(1);

        // Check result
        if ( (s.Depth() != 1)
            || (s.TopValue() != 1) ) {
            c.println("P1 test failed");
        }
        ...
    }
}
```

Independent Unit Testing

- **Factoring Out Unit Dependencies**
 - A problem with unit testing is that units are usually **interdependent**:
the method we are testing may call other methods or use other classes in the software
 - In the worst case, they all call one another
 - So how can we unit test them **individually**?

Independent Unit Testing

- **Factoring Out Unit Dependencies**
 - The test harness can provide "**stubs**" for the other units
 - The unit being tested then uses the stub method or class instead of the real thing
 - Stub methods are programmed to give "**typical**" outcomes of the real methods they take the place of, instead of a real outcome
 - This allows us to test independently of the real unit the stub replaces

Independent Unit Testing

```
// A stub for a random number generator
public int randomInt(int a, int b) {
    return 1;
}
```

Things Go Better with a Specification

- **Assertions, Please**
 - When black box unit testing, explicit **pre-** and **post-conditions** for the method we are testing help a lot
 - Pre- and post-conditions help in input and output **coverage analysis**, but more importantly, they help in automating the checking of **outcomes**
 - If an outcome meets the post condition, then we can normally accept it as correct
 - If the pre- and post-conditions are accurate enough, we can even **automate** the black box **testing process** to a large extent

Automating Black Box Unit Testing

- **Testing Tools**

- Tools such as **Jtest** and **C++test** take advantage of explicit pre-, post-, and invariant assertions in code to **automatically** do naive black box unit testing
- Given **method** and **class** interface specifications explicitly coded as **pre-**, **post-**, and **invariant** assertions written in a special, rich formal **assertion notation**, these tools can provide several benefits

```
public int ItemLength (ItemClass item) {  
    /**  
     * @pre item != null  
     * @post $result > 0  
     */  
    ...  
}
```

Automating Black Box Unit Testing

- **Testing Tools**

- Automatically generate a **test harness** for the unit
- Automatically provide **stubs** for any other units used by the method or class under test
- Automatically generate some naive **input coverage test cases** and the harness code to run them
- Automation is **naive**, so you can add your own test cases too

Automating Black Box Unit Testing

- **Problems with Testing Tools**
 - There are some limitations with unit testing tools
 1. Outcomes must be **checked by hand** (at least on the first run) since the tool cannot tell all of what was intended
 - Assertions alone are usually **insufficient**
 2. The tool can't provide **stubs** unless it knows everything that is **called** or **used** by the unit under test, that is, unless it already has the code for the unit
 - Hence, not really **black box** tests

Black Box Class Testing

- **Testing a Class Interface**
 - Naive black box **class testing** such as that done by the **JTest** and **C++Test** tools simply unit test each method of a class separately
 - In naive class testing, for each method tested,
 - the "**input**" is the current state of all class, object, and global variables as well as the parameters to the method
 - the "**output**" is the new state of all class, object, and global variables as well as the result values and exceptions of the method
 - The class specification usually has an **invariant** assertion that applies to the outcome of every method, as well as the **pre-** and **post-** assertions for each method itself

Class Traces

- Sequences of Method Calls

- Even very simple classes cannot be well specified, and hence well unit tested, simply by understanding them as a set of **independent** methods
- The **real** specification of a class often needs to reason about interdependent **sequences** of method calls, not just independent individual calls
 - For example , `stack.pop()` cannot be called before `stack.push()`
- Assertions are not well suited to specifying this kind of **sequential dependency**

Trace Specifications

- **Specifying Sequences**
 - **Trace specifications** are an explicit method for specifying the behaviour of sequences of method calls
 - Trace specifications use **trace expressions** to specify the legal sequences of method calls to an object in the class
 - A trace expression is a **sequence** of method calls with inputs and outcomes explicitly specified in the sequence

Trace Specifications

- Example of Specifying Sequences

`s.new()` , `s.Empty()` == true

`s.new()` , `s.Push(Y)` , `s.Pop()` == Y , `s.Empty()` == true

`s.new()` , `s.Push(Y)` , `s.TopValue()` == Y ,
`s.Pop()` == Y , `s.Empty()` == true

`s.new()` , `s.Push(Y)` , `s.Push(Z)` , `s.TopValue()` == Z ,
`s.Pop()` == Z , `s.Pop()` == Y , `s.Empty()` == true

Black Box Testing

Using Trace Specifications

- **Legal and Illegal Traces**
 - Trace specifications constrain the behaviour of a class using both:
 - **legal traces** (sequences that can or must happen), and
 - **illegal traces** (sequences that must never happen)
 - Trace specifications not only give us the ability to automate creation of the **test harness** and **naive** black box test cases, they also make it easy to generate black box test cases for method call **sequences** as well

Implementing Assertions

- If we use pre-, post-, and invariant **assertions** to specify our method and class interfaces, it is good practice to actually implement (check) them at **run time** on each method call
- Some languages automatically provide **assertion checking** when the assertions are given as boolean expressions in the language (which may not always be possible)

Implementing Assertions

- Checked assertions help with **every** kind of systematic testing, not just black box unit testing
- Assertions are checked every time the method or class is used, no matter what kind of testing
 - This finds errors **earlier** and pinpoints their **cause** more specifically than simply a bad outcome, because we can see exactly which assertion failed
- It is good practice to always document your specifications, assumptions, and intentions using assertions
 - Helps in **finding errors**, and helps other programmers to understand your **assumptions** and **intentions**

Black Box Integration Testing

- **Testing Subsystems**

- Black box **integration testing** applies the same ideas to testing **subsystems** integrated from smaller units
- If we began with black box **unit** testing, we can smoothly move into black box integration testing by gradually replacing each **stub** unit in a unit's test harness with the corresponding **real** unit once each has been independently unit tested
- As stubs are replaced and the **combined units** (subsystems) are re-tested at each integration, we gradually build up to the black box testing of the entire integrated system

Summary

- **Black Box Unit Testing**

- Black box testing can also be applied to **unit testing** of individual methods and classes
- **Test harnesses** are special programs written to exercise individual units under test
- **Assertion** and **trace** specifications provide interface level specifications for black box unit testing

- **References**

- Lamb, Software Engineering: Planning for Change, Ch. 13: Trace Specifications

- **Next Time: white box** testing (Prof. Cordy)