Testing and Debugging

Comp-303 : Programming Techniques
Lecture 14

Alexandre Denault
Computer Science
McGill University
Winter 2004
Announcements . . .

- I hope everybody enjoyed their week of rest.
- Assignment 2 is due today.
- Don’t forget to drop a paper copy in the hand in box.
- Most of the midterm correction is done and I will be giving them back on Thursday.
- Last day for project interview is tomorrow.
- CSGames still needs volunteers. If you want to help out this week-end, send an email to helpus@csgames.org.
Testing terminology

- **Validation**: a process designed to increase our confidence that a program works as advertised

- **Verification**: a formal or informal argument that a program works on all possible inputs

- **Testing**: a process of running a program on a limited set of inputs and comparing the actual results with expected results

- **Debugging**: a process designed to determine why a program is not working correctly

- **Defensive programming**: the practice of writing a program in a way designed specifically to ease validation and debugging
Designing test cases

• Exhaustive testing is usually impossible
  – A program with three inputs ranging from 1 to 1000 would take 1’000’000’000 test cases.
  – With a speed of 1 test per second, it would take 31 years.

• How to define a limited set of good test cases?
  – *Black-box testing*: testing from specification without regarding implementation or internal structure.
  – *Glass-box testing*: augments black-box testing by looking at implementation.
Black-box testing

• Advantages:
  – not influenced by assumptions about implementation details
  – robust with respect to changes in implementation
  – allows observers with no internal knowledge of the program to interpret the results of the test

• Disadvantages:
  – unlikely to test all parts of a program
Testing by looking at the Specs (1)

static boolean isPrime (int x)
// EFFECTS: if x is prime returns true else returns false

• The effects clause has two cases.

• Both need to be tested.
static int search (int [ ] a, int x)
        throws NullPointerException, NotFoundException
    // EFFECTS: if a is null throws NullPointerException
    // else if x is in a returns i such that a[ i ] = x
    // else throws NotFoundException

    • We should test all 3 cases mentioned in effects clause.
static float sqrt (float x, float epsilon)
// REQUIRES: x >= 0 && 0.00001 < epsilon < 0.001
// EFFECTS: returns sq such that x - epsilon <= sq*sq <= x + epsilon

• The requires clause consists of two cases:
  - x = 0 && 0.00001 < epsilon < 0.001
  - x > 0 && 0.00001 < epsilon < 0.001

• Both need to be tested.

• The effects clause can be satisfied in many ways:
  - We get an exact result
  - We get a larger result
  - We get a smaller result
static void appendVector (Vector v1, Vector v2)
    throws NullPointerException
// MODIFIES: v1 and v3
// EFFECTS: If v1 or v2 is null throws NullPointerException
// else removes all elements from v2 and appends them in
// reverse order to v1

• In certain situations, you should test beyond the specification.

• For example, if I were to call the appendVector function with
  v1 == v2, I could get a serious looping error.
Testing boundary conditions

• A program should test typical input values:
  – Arrays or sets are not empty.
  – Integers are between smallest and largest values.

• Boundary conditions usually reveal:
  – Logical errors where the path to a special case is absent.
  – Conditions which cause the underlying hardware or system to raise an exception (e.g. arithmetic overflow).

• Test data should cover all combinations of largest and smallest values:
  – Epsilon close to 0.001 and 0.00001
  – Arrays of 0 and 1 element
  – Empty strings and strings of one character
Black-box test summary

- Black-box tests are based on a program’s specification, not on its implementation.
- Black-box tests remain valid if program is reimplemented.
- Black-box tests should
  - Test all paths through a specification
  - Test boundary conditions and combinations of boundary conditions
  - Sometimes, even test a little beyond the specification
Glass-box testing

- Glass-box tests complement Black-box testing by adding a test for each possible path through the program’s implementation.
  - A glass-box test set should be path-complete.
static int maxOfThree (int x, int y, int z) {
    if (x > y)
        if (x > z) return x;
        else return z;
    if (y > z) return y; else return z;
}

• There are four possible paths through this function.

• This means we need four test cases:
  - 3,2,1
  - 3,2,4
  - 1,2,1
  - 1,2,3
Beyond Path-Completeness

static int maxOfThree (int x, int y, int z) {
    return x;
}

• However, path-completeness is not sufficient.

• Here, I only have one path. This means I would only need one test (ex: 1,2,3).

• This shows that specification should be tested, not just the implementation.

• Glass-Box testing does not reveal missing paths.
Feasibility of Path-Completeness

- Sometimes, it’s not feasible to test every path.

\[
\begin{align*}
&\text{for (int } i = 1; i \leq 100; i \text{ ++)} \\
&\quad \text{for (int } j = 1; j \leq 100; j \text{ ++)} \\
&\quad \quad \text{if (Test.predicate(i*j)) ...}
\end{align*}
\]

- In this example, we have
  \[1'267'650'600'228'229'401'496'703'205'376\] paths.

- Instead, we should test a subset.
Approximating path-completeness

- Always test each branch of a conditional.
- Loops with fixed amount of iteration.
  - test 2
- Loops with variable amount of iteration.
  - test 0,1,2
- For recursive procedures,
  - test the immediate return.
  - test one recursive call.
- Don’t forget to raise all possible exceptions.
- Use the Engineer’s induction:
  One, two, three, that’s good enough for me.
static boolean palindrome (String s) throws NullPointerException {

    // EFFECTS: If s is null throws NullPointerException else
    // returns true if s reads the same forward and backward
    // e.g. "deed" and " " are both palindromes

    int low = 0;
    int high = s.length - 1;

    while (high > low) {
        if (s.charAt(low) != s.charAt(high))
            return false;
        low ++;
        high --;
    }
    return true;
}
Testing palindrome

• Black-box testing of specification:
  – s = null
  – s = ""
  – s = "a"
  – s = "deed"
  – s = "seed"

• Glass-box testing of implementation
  – NullPointerException
  – not executing loop
  – return false in first iteration
  – return true after first iteration
  – return false after second iteration
    – add case s = "asia"
  – return true after the second iteration
• Missed any cases?
  – What if s has odd size greater than one 1?
Testing polymorphic abstractions

- This is similar to testing non-polymorphic data abstractions, but one type per parameter is not enough.
- If an interface is used, extra tests for incompatibility should be added.
  - e.g. To test OrderedList, add a String and then add an incomparable type (Integer?)
- In the related subtype approach, testing one subtype of the interface is not enough.
  - e.g. Insert a String in a SumSet that uses a PolyAdder.
Testing type hierarchies

• Blackbox testing for a subtype must include the blackbox tests of the supertype.

• However, no Glassbox testing of the supertype is required.

• When testing a subtype, you should . . .
  – Test weakened preconditions.
    Cases supported by subtype but not supertype.
  – Test strengthened postconditions.
    For example, test whether elements() of SortedIntSet are sorted.
  – Test additional methods defined for subtypes.
Unit testing and Integration testing

- Unit testing: to test whether a program unit implements its specification (i.e. specification is considered correct)
- Integration testing: to test the combination of two or more units (i.e. specification may be incompatible)
- Unit testing should always precede integration testing (divide and rule).
Tools for testing

• We might need to piece of code for unit testing:
  – Test drivers: used to test a module when using code is still unimplemented
    (executes tests + compares results with expected results)
  – Stubs: used to test a module when the code used by the module is still unimplemented
    (checks arguments and environment + produces expected results)

• Regression testing: repeat all previous tests after a change is made to fix a failed test
• Testing is used to detect errors.

• Debugging is used to understand and fix errors.

• Some common sense issues:
  – debugging takes more time than programming
  – small modules reduce debugging effort
  – well-written specifications reduce debugging effort
When debugging, apply the scientific method:

1. Study the available data.
2. Formulate a hypothesis that is consistent with the data.
3. Design and run a repeatable experiment that can refute the hypothesis.
• Find the simplest input that causes the error to occur.

  e.g. for palindrome():
  "able was I ere I saw Elba" returns false
  => hypothesis 1: the procedure doesn’t work for odd-size palindromes
     "ere" returns true
  => hypothesis 2: the procedure doesn’t work with blanks
     " " returns true
  => hypothesis 3: the procedure doesn’t work with mixed upper
     and lower case characters
     "Abba" returns false => bingo!
Debugging strategies

• Trace the code by checking intermediate results.
  \(\text{System.out.println(o.toString())}\)

• An even better idea is to use an Interactive Development Environment (IDE) that allows you to inspect variables easily.

• This allows you to find the procedure where the bug occurs (which is often most of the work).

• The bug is probably not where you think it is.

• Ask yourself where the bug is not.
  Sherlock Holmes: ”If you eliminate the impossible, what remains, however improbable, is the truth”
Debugging strategies

Try the simple things first:

- reversing the order of input arguments
- looping through an array (or String or Vector) one index too far
- failing to re-initialize a variable a second time
- copying only the top level of a data structure (shallow copy - aliasing errors)
- failing to parenthesize an expression correctly
- failing to use = instead of ==
Debugging strategies

• Get someone else to help you
  In debugging you often follow the same reasoning as when you wrote the code.

• Explain the problem to someone else
  Articulating your reasoning often reveals the source.

• If all else fails, go away
  Debugging when overly tired makes you repeat the same mistakes: take a break.

• When you find a bug, think why you put it there
  This often leads you to discover new bugs.

• Don’t be in a rush to fix the bug
  Think through all the ramifications: it is better to fix a bug you understand completely than to repeatedly apply small fixes until it works.
Defensive programming

• In development, check often:
  – requirements (e.g. check if sorted before binary search)
  – conditionals (e.g. tests all cases, even those that ”can not” occur)

• In production code, disable the checks that are too inefficient by putting them in comments (so they can be reactivated easily).
Summary

• Testing is a way of validating correctness of your code.

• Black-box testing is generated from the specification. It always remains, even when implementation changes.
  – check boundary conditions
  – check each path through the specifications

• Glass-box testing complements BB-testing by testing each path in your code.
  – all branches in a conditional
  – 0,1,2 iterations
  – 0 and 1 recursive call

• Debugging allows you to find and correct errors using the scientific method.
  (analyze data, formulate hypothesis, try to disprove)
Tool of the day: JUnit

- JUnit is a regression testing framework written by Erich Gamma and Kent Beck.
- It is used by the developer who implements unit tests in Java.
- You can create unit tests by subclassing `TestCase`.
- JUnit allows you to automate the testing of all your test cases.
- More info on JUnit is available at
  
  http://www.junit.org/
Tool of the day: Jdb

- Jdb is one of the best kept secrets of Java.
- It is a demonstration of the Java Platform Debugger Architecture that provides inspection and debugging of a local or remote Java Virtual Machine.
- It works like gdb, but a little more complicated.
- Unlike gdb, it has extensive support for tracking threads.
- To use jdb, you need to compile your classes with Debug information (-g).
- You can/should find a tutorial on the web.