Chapter 10 – Testing and Debugging

Chapter Goals
► Learn techniques to test your code
► Learn to carry out unit tests
► Understand principles of test case selection and evaluation
► Become familiar with a debugger

10.1 Unit Tests
► The single most important testing tool
► Checks a single method or a set of cooperating methods
► You don’t test the complete program that you are developing; you test the classes in isolation

Unit testing
► Avoids confusion of interactions
► More startup cost, but less work in the end
► Analogy: Testing car

Our Example
► To compute the square root of \( a \) use a common algorithm:
  ▪ Guess a value \( x \) that might be somewhat close to the desired square root (\( x = a \) is ok)
  ▪ Actual square root lies between \( x \) and \( a/x \)
  ▪ Take midpoint \( (x + a/x)/2 \) as a better guess
  ▪ Repeat the procedure. Stop when two successive approximations are very close to each other
Method converges rapidly. Square root of 100: Guess #1: 50.5
Guess #2: 26.24009900990099
Guess #3: 15.025530119986813
Guess #4: 10.840434673026925
Guess #5: 10.032578510960604
Guess #6: 10.000052895642693
Guess #7: 10.000000000139897
Guess #8: 10.0

Test Harness

For each test, you provide a simple class called a test harness.

Test harness feeds parameters to the methods being tested.

Harness

What is needed to make the testing more robust?

- More tests

Limitation: Hard to replicate tests when bugs are fixed

Solution: Use test harness to repeat values

public class RootApproximatorHarness1
{
    public static void main(String[] args)
    {
        double[] testInputs = { 100, 4, 2, 1, 0.25, 0.01 };
        for (int x = 0; x < testInputs.length; x++)
        {
            RootApproximator r = new RootApproximator(testInputs[x]);
            double y = r.getRoot();
            System.out.println("square root of " + testInputs[x] + " = " + y);
        }
    }
}

10.2 Providing Test Input

Solution #1: Hardwire series of tests

No need to memorize series of tests, already taken care of. Just run tester class.

Generate Test Cases Automatically

Instead of hard-coding array of values

1. Loop through a sample range of values

for (int x = MIN; x <= MAX; x += INCREMENT)
{
    RootApproximator r = new RootApproximator(x);
    double y = r.getRoot();
    System.out.println("square root of " + x + " = " + y);
}
Generate Test Cases Automatically

- Instead of hard-coding array of values
- Use random number generator

```java
final int SAMPLES = 100;
Random generator = new Random();
for (int i = 0; i < SAMPLES; i++) {
    double x = 1000 * generator.nextDouble();
    RootApproximator r = new RootApproximator(x);
    double y = r.getRoot();
    System.out.println("square root of " + x + " = " + y);
}
```

10.3 Test Case Evaluation

- How do you know whether the output is correct?
  - Calculate correct values by hand
    E.g., for a payroll program, compute taxes manually
  - Supply test inputs for which you know the answer
    E.g., square root of 4 is 2 and square root of 100 is 10
  - Verify that the output values fulfill certain properties
    E.g., square root squared = original value

- Use an Oracle: a slow but reliable method
e.g., use Math.pow to slower calculate \( x^{1/2} \)
  (equivalent to the square root of \( x \))

What to Test for?

- Good testing requires testing good cases
  - Reduces debugging time and ensures better product

- Test all of the features of the method
  - Positive tests – normal behavior of method
  - Boundary test cases (e.g. \( x = 0 \)), make sure end conditions are correct
  - Negative cases – program should reject

```java
for (int i = 1; i <= SAMPLES; i++) {
    double x = 1000 * generator.nextDouble();
    RootApproximator r = new RootApproximator(x);
    double y = r.getRoot();
    if (Numeric.approxEqual(y * y, oracleValue)) {
        System.out.print("Test passed: ");
        passcount++;
    } else {
        System.out.print("Test failed: ");
        failcount++;
        System.out.println("x = " + x + ", root squared = " + y * y);
    }
}
```
10.4 Regression Testing and Test Coverage

► Save test cases
   • Particularly tests that reveal bugs

► Use saved test cases in subsequent versions

► A test suite is a set of tests for repeated testing

Why keep a test case?

► Very common for bugs to show up later
   • We often think we fixed a bug, but just covered it up

► Cycling = bug that is fixed but reappears in later versions

► Regression testing: repeating previous tests to ensure that known failures of prior versions do not appear in new versions

Test Coverage

► Test coverage: measure of how many parts of a program have been tested
   • Good testing tests all parts of the program
   • E.g. every line of code is executed in at least one test

► Example: Test all possible branches inside of an if statement

Testing Levels

► The part of the program that is being tested.
   • Unit
   • System

Unit Level Testing

► Test individual units (classes) in isolation.

► Test features of the unit.

► Multiple test cases for each feature.

System Level Testing

► Test interactions of the various parts of the application.

► Completed after the unit tests pass.
Test Coverage

- **Black-box testing**: test functionality without consideration of internal structure of implementation
  - Useful since this is what the user interacts with
- **White-box testing**: take internal structure into account when designing tests
  - Unit testing
  - Why? Can not prove absence of bugs, only presence

Black Box Testing

- a.k.a. behavioral, functional, closed
- Test against the specification
- does not need the program source
- Internal implementation is unknown

White Box Testing

- a.k.a. structural, clear box, open
- Tests against the implementation
- Every line of source code is executed at least once.
- Tests exception handling code.

White Box Testing

- Advantages
  - User's point of view
  - Less biased
  - Can discover if part of the specification has not been fulfilled.
- Disadvantages
  - May be redundant
  - Can be difficult to design.
  - Testing every possible input stream is unrealistic

"Fully" Tested

- To fully test a software product, the following are required:
  - Unit testing and system level testing
  - Black and white box testing
- Does this ensure there are no bugs?
10.6 Debugger

► To avoid/find problems, most professionals use a debugger

► Programs rarely (never) run perfectly the first time
  ▪ Think of perfect first draft to paper

► The larger your programs, the harder it is to debug them simply by printing out values or looking at the code

Debugger - program to run your program and analyze its run-time behavior

Debuggers can be part of your IDE (Eclipse, BlueJ) or separate programs (JSwat)

Three key concepts:
  ▪ Breakpoints
  ▪ Stepping
  ▪ Inspecting variables

Breakpoint

► Breakpoint – the debugger stops running the program when it hits a
  ▪ Setup by programmer

► Once stopped, you can see the state of all the variables

Stepping

► Once a breakpoint is hit, two options
  ▪ Step through the next few statements carefully inspecting
    ▪ Slow, but useful in heavy calculations
  ▪ Single-step – line by line execution
  ▪ Step into – doesn’t just skip line to line, but goes into method calls within each line
  ▪ Step over – Execute the method and stop
    ▪ Run at full speed until the next breakpoint
Example

► Current line:

```java
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in "+ input + ": "+ syllables);
```

Step Over

► Next step:

```java
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in "+ input + ": "+ syllables);
```

Step into

```java
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    ...

Which to choose

► If the method is suspect (may be cause of problem) – step into

► If you are sure the method works correctly – step over

What if a test fails?

► It’s time to debug!

► We will want (need) a strategy.

► Debugging Strategy
  • A systematic way to find and fix bugs.

Debug Strategy 1: Trial & Error

1. Write code
2. Run program
3. Fix bugs

► Advantages
  ▪ no planning required

► Disadvantages
  ▪ Spend more time trying to find and fix bugs than you did writing the program.
Debug Strategy 2:
Trial & Error with a Debugger
1. Write code
2. Run program
3. Use Debugger program to fix bugs

► Advantages
  - no planning required

► Disadvantages
  - Spend lots of time in the debugger program trying to find and fix the bugs (but maybe less than the first strategy)

Debug Strategy 3:
Incremental Development with Testing
1. Write small amount of code with testing and debugging in mind.
2. Document and write tests for each new piece of code.
3. Run tests on new code
4. Fix any bugs before adding new code.

General Debugging Strategy
1. Identify the bug. Must be repeatable and testable.
2. Write test(s) that fails for the bug.
3. Locate the source of the bug. HOW?
4. Understand the problem before you fix it.
5. Try your fix and run all tests.
6. Repeat until all tests pass.

How do we find the source of the bug?
► Pick a point to start looking
  - Beginning (main method)
  - Divide and conquer

► Trace the code.
  (Look for variables with unexpected values)
  - Manual trace
  - Trace messages
  - System.out.println(…)
  - Use java.util.logging.Logger class
  - Debugger program