Software Testing II

Assessment 2

Lab: Queue

MSc Projects

Common Errors

- Can be from a particular programming community.
- Well-instrum and summarise error occurr
- Professional good practice's you sensitive to the errors you make personally.
- The following are the "top three" from David Reilly's top ten Java programming errors
  - Concurrent access to shared variables by threads (3)
  - Capitalization errors (2)
  - Null pointers (1)

Concurrent access to shared variables by threads

```
public class MyCounter {
  private int count = 0; // count starts at zero

  public void in
    count = coun
  }

  public int getCount() {
    return count;
  }
}
...
                MyCounter c;
// Thread 1                    // Thread 2
c.incCount(1);                 c.incCount(1);
                // join
                c.getCount() == ?
```

Concurrent access to shared variables by threads


```
public class MyCounter {
  private int co

  public synchronized void incCoun
    count = count + amount;
  }

  public int getCount() {
    return count;
  }
}
```

Capitalization Errors

- All methods and member variables in the Java API begin with lower

- All methods and member capitalization where a new word begins, such as doubleValue().

Null pointer

```
public static void main(String args[]) {
  String[] list = new String[3]; // Accept up to 3 parameters
  int index = 0;

  while( (index < args.l
    list[index] = args
    index++;
  }

  // Check all the parameters
  for(int i = 0; i < list.length; i++) {
    if(list[i].equals("-help")) {
      // .........
    } else if(list[i].equals("-cp")) {
      // .........
    }
    // [else .....]
  }
}
```

Test Coverage

- Statement Coverage
- Branch Co
- Condition Coverage

Statement Testing

- **Statement Adequacy**: all statements have been executed

- **Statement Coverage:** for a st T, this is the quotient of the number of executed during a run of T (not counting repe number of statements in the program.

Running example

Running example

- A = [-1], X=2

Branch Coverage

- **Branch adequacy**:
  Let T be a t T satisfies the branch adequacy n B of P there exists at least one test case that

- **The branch coverage** for a he ratio of branches tested by the suite and the number of branches in the program under test.

Running example

- A = [-1], X=2

Running example

- A = [-1], X=2
- A = [-1,1], X=2

Condition coverage

- **Condition adequacy** criterion is:
  Let T be a t overs all the basic conditions in P. evaluates to true under some test in T a s to false under some test in T.

Running example

- A = [-1], X=2
- A = [-1,1], X=2

Running example

- A = [-1], X=2
- A = [-1,1], X=2
- A = [-1,1,2], X = 2

Mutation Testing

- What is a mutation?
- What is mu
- When should we use mutat
- Examples

What is a mutation?

- A mutation is a small change in a program.
- Such small model low level defects that arise in the pr ing systems
- Ideally mutations should m el defect creation.

What is Mutation Testing?

- Mutation testing is a structural testing method aimed at assessing/i est suites, and estimating t in systems under test.

- The process, given program ite T, is as follows:

  - We systematically apply mu rogram P to obtain a sequence P1, P2,... Pn of mutants of P. Each mutant is derived by applying a single mutation operation to P.
  - We run the test suite T on each of the mutants, T is said to kill mutant Pj if it detects an error.
  - If we kill k out of n mutants the adequacy of T is measured by the quotient
  - k/n. T is mutation adequate if k = n.

When should we use mutation testing?

- Structural test suites are directed at identifying defects in the code. One goal of mutation testing is to assess or improve the efficacy of test suites in discovering defects.
- When we are ing we are worried about defects then we are keen to measure the **R** **D)** in the program P under test.
- The Residual Defect Density is usually measured in defects per thousand lines of code.

Using Mutation Testing to Estimate the RDD

- Suppose we have an estimate r of the RDD of programs produced by our development process for the same subject to test (this could be gathered using production data and field experience, or it could be based on how many tests have already detected).
- Generate n mutants of the progra
- Test each mutant with the test suite T .
- Find the number, k, of mutants that are killed by T. To yield a non-zero RDD. we need to test enough mutants to ensure that $0 < k < n$.
- Use r. $(n - k)/k$ as the estimate for the RDD of the tested program.
- k/n is a measure of the adequacy of T in finding defects in P

Kinds of Mutation

- **Value Mutations**: these mutations involve changing the values of constants or parameters (by adding or subtracting values etc), e.g. loop bounds – being off by one or six is a very common error.
- **Decision Mutations**: this involves modifications to reflect potential slips and errors in the coding of conditions in programs, e.g. a typical mutation might be replacing a < in a comparison
- **Statement Mutations**: these might involve deleting certain lines to reflect omissions in coding or swapping the order of lines of code. There are other operations, e.g. changing operations in arithmetic expressions. A typical omission might be to omit the increment on some variable in a while loop.

Offutt's Mutations for Inter-Class Testing

Value Mutation

Decision Mutation

Statement Mutation

Observations

- Mutations model low level errors in the mechanical production process. Modelling design errors is much harder because they involve large changes to collocated code throughout the program.
- Ensuring test sets s often enough to ensure they kill mutants (because n do not "make sense" and so provoke a failure if executed).
- Black-box test sets are poorer at killing mutants – we'd expect this because black-box tests are driven more by the operational profile than by the need to cover statements.
- We could see mutation testing as a way of forcing more diversity on the development of test sets if we use a black-box approach as our primary test development approach.

Regression Testing

- Regression testing is applied to code immediately after changes a
- The goal is to ass not had unintended consequences o bject re made.
- We can apply regression testing d pment and in the field after the system has been up ained in some other way.
- Give us confidence that we can change the object of test while maintaining its intended behaviour.
- • Regression testing is an important way of monitoring the effects of change.

More system testing

- Capacity Testing
- Stress Testing
- Usability Testing
- Security Testing
- Performance Testing
- Reliability Testing
- Availability Testing
- Documentation Testing
- Configuration Testing

PI 8.1

```
if(a || b)) {
    test1 = true;
} else {
    if(c) {
            test2 = t
        }
}
```


How many of the following statements is correct?

For full statement coverage, we need a=true b=false; a=false b=false c=true

For full branch coverage, we need a=true b e b=false c=true; a=false e

For full condition coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false; a=false, b=true

A: 0
B: 1
C: 2
D: 3

PI 8.1

```
if(a || b)) {
    test1 = true;
} else {
    if(c) {
            test2 = t
        }
}
```


How many of the following statements is correct?

For full statement coverage, we need a=true b=false; a=false b=false c=true

For full branch coverage, we need a=true b e b=false c=true; a=false e

For full condition coverage, we need a=true b=false; a=false b=false c=true; a=false b=false c=false; a=false, b=true

A: 0
B: 1
C: 2
D: 3