White Box Testing

CS 240 – Advanced Programming Concepts

White Box Testing

- Looking at the class's internal implementation, in addition to its inputs and expected outputs, enables you to test it more thoroughly

- Testing that is based both on expected external behavior and knowledge of internal implementation is called "white box testing"

White Box Testing

- White box testing is primarily used during unit testing
- Unit testing is usually performed by the engineer who wrote the code
- In rare cases an independent tester might do unit testing on your code

Complete Path Coverage

- Test ALL possible paths through a subroutine
- What test cases are needed to achieve complete path coverage of this method? (white-box-example.java)
- Some paths may be impossible to achieve. Skip those paths ☺
- Often there are too many paths to test them all, especially if there are loops in the code. In this case, we use less complete approaches:
  - Line coverage
  - Branch coverage
  - Condition testing
  - Loop testing

Line Coverage

- At a minimum, every line of code should be executed by at least one test case
- What test cases are needed to achieve complete line coverage of this subroutine? (white-box-example.java)
- Developers tend to significantly overestimate the level of line coverage achieved by their tests
- Coverage tools (like Jacoco) are important for getting a realistic sense of how completely your tests cover the code
- Complete line coverage is necessary, but not sufficient

Branch Coverage

- Similar to line coverage, but stronger
- Test every branch in all possible directions
- If statements
  - test both positive and negative directions
- Switch statements
  - test every branch
  - If no default case, test a value that doesn't match any case
- Loop statements
  - test for both 0 and > 0 iterations

Branch Coverage

- What test cases are needed to achieve complete branch coverage of this subroutine? (white-box-example.java)

- Why isn't branch coverage the same thing as line coverage?

Branch Coverage

- What test cases are needed to achieve complete branch coverage of this subroutine? (white-box-example.java)

- Why isn't branch coverage the same thing as line coverage?
  - Consider an if with no else, or a switch with no default case
  - Line coverage can be achieved without achieving branch coverage

Complete Condition Testing

- For each compound condition, C
- Find the simple sub-expressions that make up C
  - Simple pieces with no ANDs or ORs
  - Suppose there are n of them
- Create a test case for all 2^n T/F combinations of the simple sub-expressions
  - if (!done && (value < 100 || c == 'X')) …
  - Simple sub-expressions
    - !done, value < 100, c == 'X'
    - n = 3
    - Need 8 test cases to test all possibilities

Complete Condition Testing

- Use a "truth table" to make sure that all possible combinations are covered by your test cases
- Doing this kind of exhaustive condition testing everywhere is usually not feasible

	!done	value < 100	c == 'X'
Case 1:	False	False	False
Case 2:	True	False	False
Case 3:	False	True	False
Case 4:	False	False	True
Case 5:	True	True	False
Case 6:	True	False	True
Case 7:	False	True	True
Case 8:	True	True	True

Partial Condition Testing

- A partial, more feasible approach
- Identify a subset of test cases that meet the following criteria:
  1. For conditions that can independently affect the overall result, identify a true case and a false case for that condition where all other values are the same and the overall result is different
  2. For conditions that cannot independently affect the overall result, identify a true case and a false case where the following criteria are met:
     1. All other subexpressions produce the same result (the subexpression the condition is part of affects the result)
     2. The condition is the only part of the subexpression that changes
     3. The overall result is different
- One test case may cover more than one of these, thus reducing the number of required test cases

Partial condition Testing Truth Table

if (!done && (value < 100 || c == 'X')) …

	!done	value < 100	c == ‘X’	(value < 100 || c == ‘X’)	Overall Result
Case 1:	False	False	False	False	False
Case 2:	True	False	False	False	False
Case 3:	False	True	False	True	False
Case 4:	False	False	True	True	False
Case 5:	True	True	False	True	True
Case 6:	True	False	True	True	True
Case 7:	False	True	True	True	False
Case 8:	True	True	True	True	True

Partial condition Testing

if (!done && (value < 100 || c == 'X')) …

- ‘!done’ can independently affect the overall result
  - Identify a true case and a false case for that condition where all other values are the same and the overall result is different

	!done	value < 100	c == ‘X’	(value < 100 || c == ‘X’)	Overall Result
Case 1:	False	False	False	False	False
Case 2:	True	False	False	False	False
Case 3:	False	True	False	True	False
Case 4:	False	False	True	True	False
Case 5:	True	True	False	True	True
Case 6:	True	False	True	True	True
Case 7:	**False**	True	True	True	**False**
Case 8:	**True**	True	True	True	**True**

Partial condition Testing

if (!done && (value < 100 || c == 'X')) …

- 'value < 100' cannot independently affect the overall result
  - Identify a true case and a false case for that condition where all other subexpressions are the same, the other parts of the subexpression are the same and the overall result is different

	!done	value < 100	c == 'X'	(value < 100 || c == 'X')	Overall Result
Case 1:	False	False	False	False	False
Case 2:	True	**False**	False	**False**	**False**
Case 3:	False	True	False	True	False
Case 4:	False	False	True	True	False
Case 5:	True	**True**	False	**True**	**True**
Case 6:	True	False	True	True	True
Case 7:	False	True	True	True	False
Case 8:	True	True	True	True	True

Partial condition Testing

if (!done && (value < 100 || c == 'X')) …

- ‘c == ‘X’’cannot independently affect the overall result
  - Identify a true case and a false case for that condition where all other subexpressions are the same, the other parts of the subexpression are the same and the overall result is different

	!done	value < 100	c == ‘X’	(value < 100 || c == ‘X’)	Overall Result
Case 1:	False	False	False	False	False
Case 2:	True	False	**False**	**False**	**False**
Case 3:	False	True	False	True	False
Case 4:	False	False	True	True	False
Case 5:	True	True	False	True	True
Case 6:	True	False	**True**	**True**	**True**
Case 7:	False	True	True	True	False
Case 8:	True	True	True	True	True

What test cases do we need to achieve

```
// Compute Net Pay
totalWithholdings = 0;

for ( id = 0; id < numEmployees; ++id) {

    // compute social security withholding, if below the maximum
    if ( m_employee[ id ].governmentRetirementWithheld < MAX_GOVT_RETIREMENT) {
         governmentRetirement = ComputeGovernmentRetirement( m_employee[ id ] );
    }

    // set default to no retirement contribution
    companyRetirement = 0;

    // determine discretionary employee retirement contribution
    if ( m_employee[ id ].WantsRetirement && EligibleForRetirement( m_employee[ id ] ) ) {
        companyRetirement = GetRetirement( m_employee[ id ] );
    }

    grossPay = ComputeGrossPay( m_employee[ id ] );

    // determine IRA contribution
    personalRetirement = 0;
    if (EligibleForPersonalRetirement( m_employee[ id ] ) {
        personalRetirement = PersonalRetirementContribution( m_employee[ id ], companyRetirement, grossPay );
    }

    // make weekly paycheck
    withholding = ComputeWithholding( m_employee[ id ] );
    netPay = grossPay - withholding - companyRetirement - governmentRetirement - personalRetirement;
    PayEmployee( m_employee[ id ], netPay );

    // add this employee's paycheck to total for accounting
    totalWithholdings += withholding;
    totalGovernmentRetirement += governmentRetirement;
    totalRetirement += companyRetirement;
}

SavePayRecords( totalWithholdings, totalGovernmentRetirement, totalRetirement );
```

Line coverage?

Branch coverage?

Complete condition testing?

Partial condition testing?

Loop Testing

- Design test cases based on looping structure of the routine

- Testing loops
  - Skip loop entirely
  - One pass
  - Two passes
  - N-1, N, and N+1 passes [N is the maximum number of passes]
  - M passes, where $2 < M < N-1$

Loop Testing

```
public int readInputStreamAsChars(InputStream is, char[] buffer) throws IOException {
    int count = 0;
    while (count < buffer.length) {
        int c = is.read();

        if (c == -1 || c == '\n') {
            break;
        } else {
            buffer[count++] = (char) c;
        }
    }

    return count;
}
```

What test cases do we need?

1) Skip loop entirely:
   a. buffer.length == 0
2) Exactly one pass:
   a. Empty input stream OR buffer.length == 1
3) Exactly two passes:
   a. Input stream length 1 OR buffer.length == 2
4) N-1, N, and N+1 passes:
   a. buffer.length = **stream length -2**, stream length, and stream length + 1
5) M passes, where 2 < M < N-1
   a. line of length buffer.length / 2 where buffer.length >= 3

Data Flow Testing

- The techniques discussed so far have all been based on "control flow"
- You can also design test cases based on "data flow“ (i.e., how data flows through the code)
- Some statements "define" a variable’s value (i.e., a “variable definition”)
  - Variable declarations with initial values
  - Assignments
  - Incoming parameter values
- Some statements "use" variable’s value (i.e., a “variable use”)
  - Expressions on right side of assignment
  - Boolean condition expressions
  - Parameter expressions

Data Flow Testing

- For every "use" of a variable
  - Determine all possible places in the program where the variable could have been defined (i.e., given its most recent value)
  - Create a test case for each possible (Definition, Use) pair

Data Flow Testing

```
if ( Condition 1 ) {
    x = a;
} else {
    x = b;
}

if ( Condition 2 ) {
    y = x + 1;
} else {
    y = x - 1;
}
```

What test cases do we need?

Definitions:

- x = a;
- x = b;

Uses:

- y = x + 1;
- y = x – 1;

1. (x = a, y = x + 1)
2. (x = b, y = x + 1)
3. (x = a, y = x – 1)
4. (x = b, y = x – 1)

Data Flow Testing

- Use data flow testing to design a set of test cases for this method: data-flow-example.java

Relational Condition Testing

- Testing relational sub-expressions
- (E1 op E2)
- ==, !=, <, <=, >, >=

- Three test cases to try:
  - Test E1 == E2
  - Test E1 slightly bigger than E2
  - Test E1 slightly smaller than E2

Internal Boundary Testing

- Look for boundary conditions in the code, and create test cases for boundary – 1, boundary, boundary + 1

```
void sort(int[] data) {
    if (data.length < 30) {
        insertionSort(data);
    } else {
        quickSort(data);
    }
}
```

Internal Boundary Testing

```
const int CHUNK_SIZE = 100;

char * ReadLine(istream & is) {
        int c = is.get();
        if (c == -1) {
                return 0;
        }

        char * buf = new char[CHUNK_SIZE];
        int bufSize = CHUNK_SIZE;
        int strSize = 0;

        while (c != '\n' && c != -1) {
                if (strSize == bufSize - 1) {
                        buf = Grow(buf, bufSize);
                        bufSize += CHUNK_SIZE;
                }

                buf[strSize++] = (char)c;

                c = is.get();
        }

        buf[strSize] = '\0';

        return buf;
}
```

What test cases do we need?

Lines of length 99, 100, 101

Data Type Errors

- Scan the code for data type-related errors such as:
  - Arithmetic overflow
    - If two numbers are multiplied together, what happens if they're both large positive values? Large negative values?
    - Is divide-by-zero possible?
  - Other kinds of overflow
    - If two strings are concatenated together, what happens if they're both unusually long
  - Casting a larger numeric data type to a smaller one
    - short s = (short) x; // x is an int
  - Combined signed/unsigned arithmetic

Built-in Assumptions

- Scan the code for built-in assumptions that may be incorrect
  - Year begins with 19
  - Age is less than 100
  - String is non-empty
  - Protocol in URL is all lower-case
    - What about "hTtP://..." or FTP://...?

Limitations of White Box Testing

- Whatever blind spots you had when writing the code will carry over into your white box testing
  - Testing by independent test group is also necessary
- Developers often test with the intent to prove that the code works rather than proving that it doesn't work
- Developers tend to skip the more sophisticated types of white box tests (e.g., condition testing, data flow testing, loop testing, etc.), relying mostly on line coverage
- White box testing focuses on testing the code that's there. If something is missing (e.g., you forgot to handle a particular case), white box testing might not help you.
- There are many kinds of errors that white box testing won't find
  - Missing functionality
  - Timing and concurrency bugs
  - Performance problems
  - Usability problems
  - Etc.