Software Testing

Comp 30160: Object Oriented Design

- We test to evaluate properties of software, e.g.

  - Correctness

  - Reliability

  - Performance

  - Memory Usage

  - Security

  - Usability

  - ...

- Our focus will be testing for **correctness**.
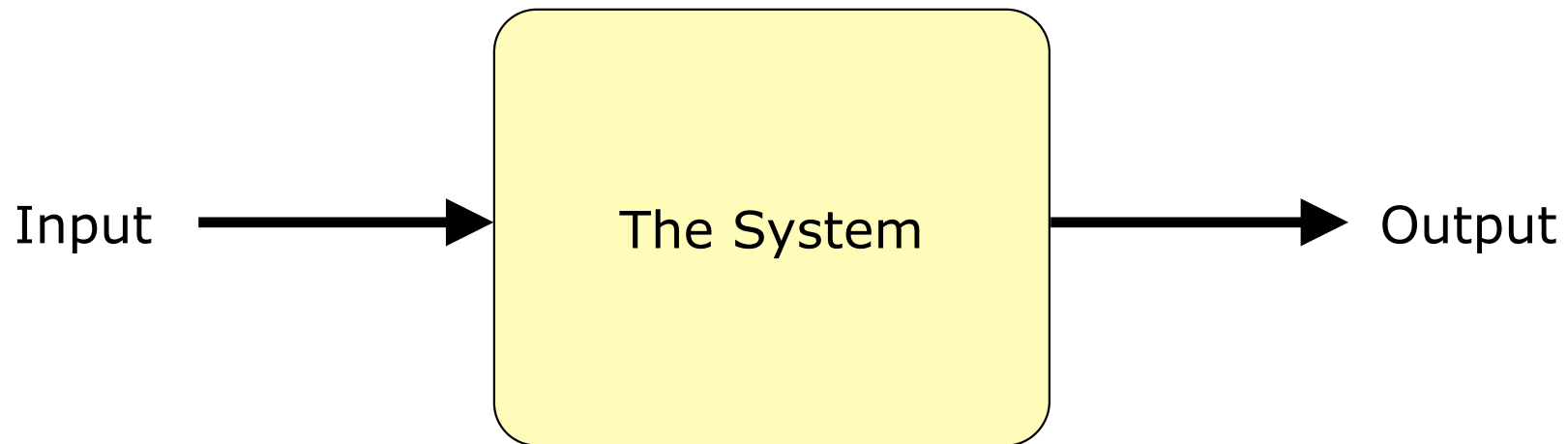
  - "Code does what it should"

Types of Testing

- User acceptance testing

- Integration testing

- Alpha testing, beta testing

- Black box testing, white box testing

- Unit testing

- Our focus will be on **unit testing**.

  - Testing classes/methods on their own

- We look at some theory first, then look at tool support for unit testing (JUnit) and at **Test-Driven Development** (TDD).

Roadmap of this Topic

- Introduction to Testing

- Black-box Testing

- White-box testing:

    - statement coverage
    - path coverage
    - branch testing
    - condition coverage
    - Cyclomatic complexity
- Summary

How do you test a system?

- Input test data to the system.

- Observe the output:
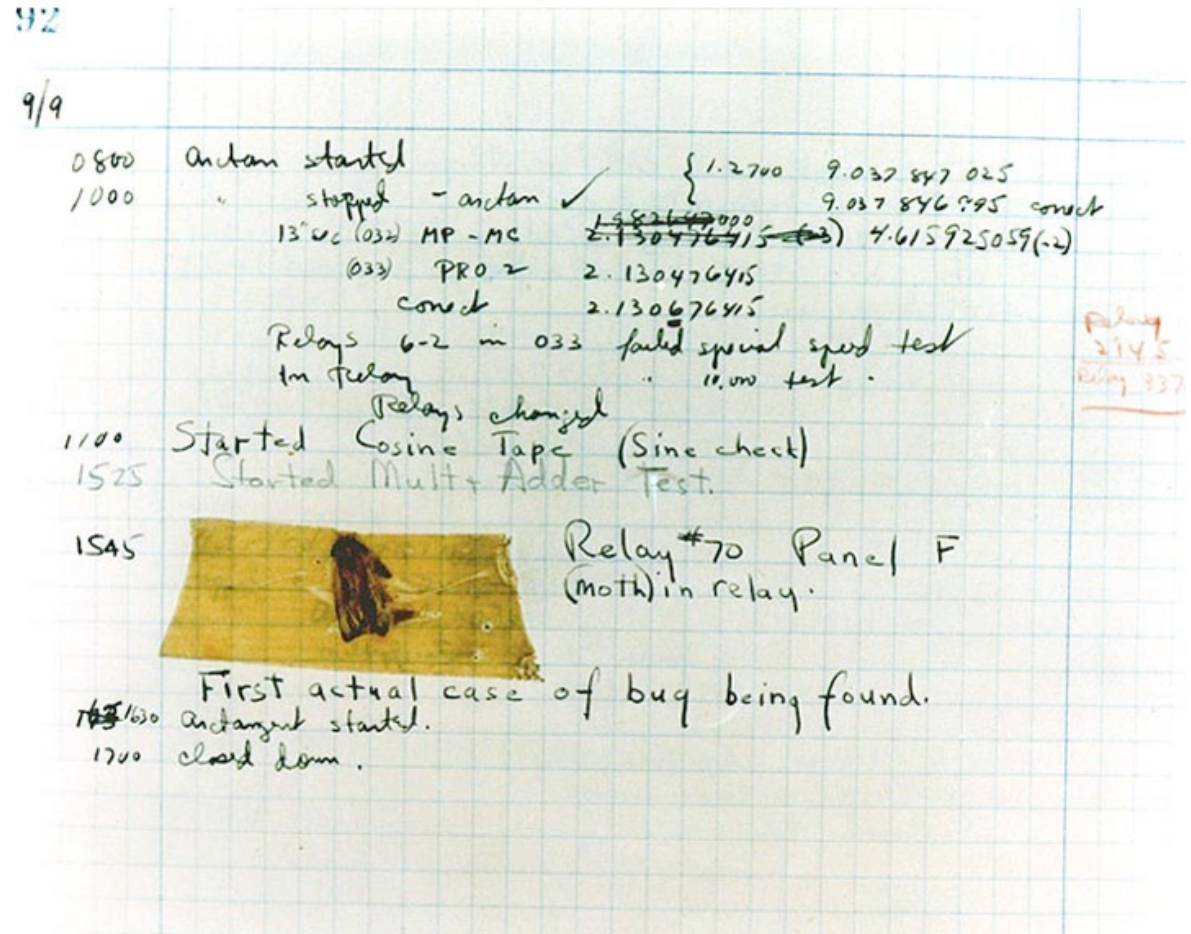
  - Check if the system behaved as expected.

Input ➤ **The System** ➤ Output

- If the program does not behave as expected:

  - note the conditions under which it failed.

  - debug and correct.

Bugs and Failures

- A **bug** (aka **error, defect**) is a mistake in the code.

- A **failure** is a manifestation of an error.

- Presence of a bug may not lead to a failure.

Photo # NH 96566-KN (Color)   First Computer "Bug", 1947

hold

stop

Test cases and Test suite

- Test software using a set of carefully designed **test cases**:

  - the set of all test cases is called the **test suite**

- A test case is a triplet [I,S,O]:

  - I is the data to be input to the system,
  - S is the state of the system at which the data is input,
  - O is the expected output from the system.

- How do we know what the output should be?

  - Perhaps we need an **oracle**.

- Does it ever make sense to test when we don't know what the outputs should be?   Yes!

hold

stop

Regression Testing

- To regress is to 'return to a former or less developed state.'

  - a bad thing

- Regression in software is where updates to the software lead to the accidental introduction of new errors (or re-introduction of old bugs)

  - so the software gets worse in some sense

- Regression testing simply involves testing if the software has regressed

  - so creating test cases without knowing what the correct output should be can be very useful
  - it enables regression testing to take place at a later stage

Exhaustive Testing

- Exhaustive testing of any non-trivial system is impractical

  - input data domain is extremely large.

- How long would it take to test this simple method **on all possible inputs?** Assume 32-bit integers and that we can do 1,000,000 tests per second:

```
/**
 * Return sum of arguments
 **/
int sum(int x, int y) {
    ...
}
```

Design of Test Cases

- If test cases are selected randomly

    - many test cases will not detect errors not already detected by other test cases in the suite => waste of time running and maintaining useless tests

- The number of test cases in a test suite is **not** a good indication of the effectiveness of the testing.

- Design an optimal test suite:

    - of reasonable size
    - to uncover as many errors as possible.
    - Bigger does not mean better!

hold

stop

Test Suite Examples

- Consider this method and two possible test suites for it

```
/**
 * Return maximum of arguments
 **/
int maxInt(int x, int y) {
    ...
}
```

- Test suite I: `{(x=3,y=2);(x=2,y=3)}`

- Test suite II `{(x=3,y=2);(x=4,y=3); (x=5,y=1)}`

hold

stop

- Now consider this buggy implementation and decide which test suite is better:

```
/**
* Return maximum of arguments
**/
int maxInt(int x, int y) {
   int max;
   if (y > x)
      max = y;
   max = x;
   return max;
}
```

- Test suite 1: {(x=3,y=2);(x=2,y=3)}

- Test suite II {(x=3,y=2);(x=4,y=3); (x=5,y=1)}

Design of Test Cases

- Systematic approaches are required to design an optimal test suite:

  - each test case in the suite should detect different errors.

- Two main approaches to designing test cases:

  - **Black-box** approach

  - **White-box** approach

From here on, we assume we're testing a single **method**.

Black-box/White-box Testing

- In **black-box testing**, test cases are designed using only a functional specification of the software under test (**SUT**), without any knowledge of the internal structure of the software.

  - Also known as **functional testing**.

- White-box testing involves examining the code itself.

  - requires knowledge about the internal structure of the software under test.

  - Also known as **structural testing**.

Black-box Testing

- Two basic approaches to design black box test cases:

  - **Equivalence class partitioning**
  - **Boundary value analysis**

- (Two fancy names for two very simple concepts)

Simple Income Tax Example

- Consider this specification of a simple income tax system:

Income Tax System	
Salary	**Tax**
€0 - 20K	15% of total income
€20K - 50K	€3K + 25% of amount over €20K
Above €50K	€10.5K + 40% of amount over €50K

The input clearly forms three **equivalence classes**:

> €0 - 20K
> €20K -50K
> €50K+

and at least one case from each class should be tested. Testing many cases from the same equivalence class is probably a waste of time.

Boundary value analysis

- Boundaries are areas where errors frequently lie:

    - Iterating once too many or once too few

    - Overunning an array, or not processing the last element

    - Date errors, not considering zero or negative values...

In the previous example, there are clearly three boundaries to be tested explicitly:

€0
€20,000
€50,000

White-Box Testing

- The distinguishing feature of white box testing is that it's based on examining the code itself.

- A key feature is to what extent the testing **covers** the code. Several approaches to code coverage exist:

  - statement coverage
  - branch coverage
  - condition coverage
  - path coverage
  - mutation testing
  - data flow-based testing

Statement Coverage

- Statement coverage:

  - design test cases so that every statement in a method is executed at least once.

```
int foo(int x, int y) {
   int z=0;
   if ((x>0) && (y>0)) {
      z = x;
   }
   return z;
}
```

- Testing with {(x=1, y=1)} gives 100% statement coverage.

Statement coverage criterion

- Based on the observation that an error in a method cannot be discovered unless the part of the method containing the error is executed.

- However, observing that a statement behaves properly for one input value provides no guarantee that it will behave correctly for all input values.

Example

```
int gcd(int x, int y) {
1.    while (x != y) {
2.        if (x>y)
3.            x = x-y;
4.        else y = y-x;
5.    }
6.    return x;
}
```

- How can we achieve 100% statement coverage?

- E.g. by choosing the test set {(x=4,y=3), (x=3,y=4)}, 100% statement coverage is achieved.

Branch Coverage

- Branch coverage means testing each branch of each control structure, e.g. in `if`, `case`, `while` statements

- Test cases are designed such that different branch conditions are given true and false values.

- Branch testing guarantees statement coverage, it's a **stronger** testing criterion compared to statement coverage-based testing.

```
int foo(int x, int y) {
   int z=0;
   if ((x>0) && (y>0)) {
      z = x;
   }
   return z;
}
```

{(2,2)} gives statement coverage but not branch coverage

hold

stop

Branch Coverage Example

```
   int gcd(int x, int y) {
1.    while (x != y) {
2.       if (x>y)
3.          x = x-y;
4.       else y = y-x;
5.    }
6.    return x;
   }
```

- How can we achieve 100% branch coverage?

- Possible test suite that gives **branch coverage** would be:
  {(x=3,y=3),(x=3,y=2), (x=3,y=4)}

Condition Coverage

- In **basic condition coverage** test cases are designed such that each boolean sub-expression of a composite conditional expression is given both true and false values.

  - Slightly more complex in the presence of "short circuit" evaluation of C++/Java, but we pass over that for simplicity

- In **complete condition coverage** test cases are designed such that all possible combinations of boolean values are tested,

  - requires 2^n test cases for n boolean variables

  - very demanding criterion

- **Modified condition/decision coverage** is an "in between" technique, better than basic condition coverage but not as demanding as multiple condition coverage.

Modified condition/decision coverage

- Modified condition/decision coverage requires

  - basic condition coverage is satisfied, and

  - **each condition must affect the outcome independently.**

- See example…

Modified condition/decision coverage

- Consider the expression: `if ((a || b) && c)`

  - { (a=true, b=true, c=true), (a=false, b=false c=false)} yields **basic condition coverage**

  - **complete condition coverage** requires 8 test cases

- What testcases are sufficient to yield modified condition/decision coverage?

a	b	c	(a||b)&&c
FALSE	FALSE	TRUE	FALSE
TRUE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	TRUE
TRUE	TRUE	FALSE	FALSE

Path Coverage

- If a method has a loop, testing **all paths** through the method is impossible

- **Basis path coverage** means

    1. creating a maximum set of **linearly independent paths** for the method being tested
    2. designing test cases such that all **linearly independent paths** are executed at least once.

- Defined in terms of the **control flow graph** (CFG) of a method.

- A CFG describes:

    - the possible sequences in which different statements of a method get executed ("flow of control").

hold

stop

How to draw a Control Flow Graph?

- Number all the statements of a method.

- Numbered statements represent nodes of the control flow graph

- An edge from one node to another node exists:

  - if execution of the statement representing the first node can result in transfer of control to the other node.

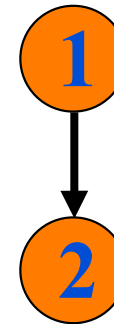What follows can all be expressed in formal graph theory. We take an informal approach.

hold

stop

```
    int gcd(int x, int y) {
1.    while (x != y) {
2.       if (x>y)
3.          x = x-y;
4.       else y = y-x;
5.    }
6.    return x;
    }
```

Sequence:

```
1   a=5;

2   b=a*b-1;
```

Selection:

```
1  if(a>b) then
2     c=3;
3  else c=5;
4     c = c/2;
```

Iteration:

```
1   while(a>b) {
2       b=b*a;
3   }
4   x=10;
```

Paths

- A **path** through a program/method is a node and edge sequence from the starting node to a terminal node of the CFG.

  - There may be several terminal nodes for a method.

- A **linearly independent path** is any path through the method/program that introduces at least one new **edge*** that is not included in any other independent path.

  - Can be hard to identify for a complicated method
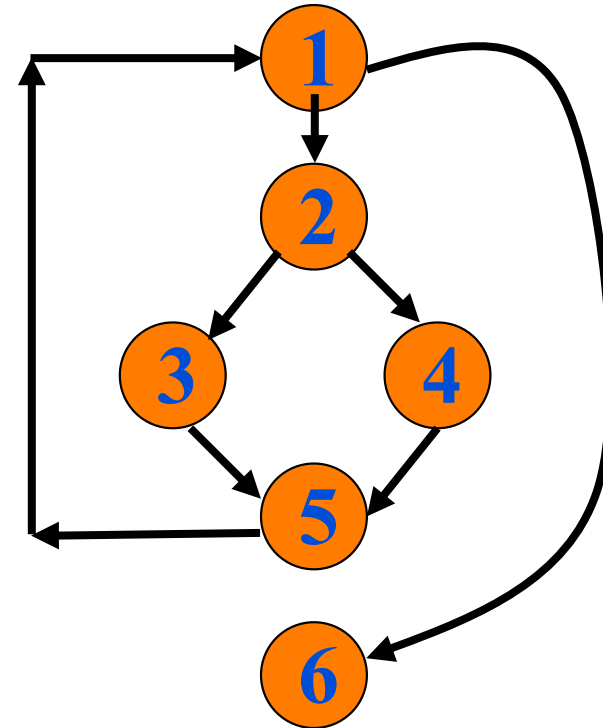 - (in fact theoretically impossible in the general case)

* If a new node is introduced, that implies that a new edge has been introduced as well. However, a new edge can be introduced without there being a new node involved.

McCabe's Cyclomatic Metric

- Provides an upper bound for the number of linearly independent paths of a method

  - Some paths may actually be impossible to traverse, hence it's an upper bound

- Provides a practical way of determining the maximum number of linearly independent paths in a method.

hold

stop

McCabe's cyclomatic metric

- Given a control flow graph G, cyclomatic complexity V(G):
  - V(G)= E-N+2
    - N is the number of nodes in G
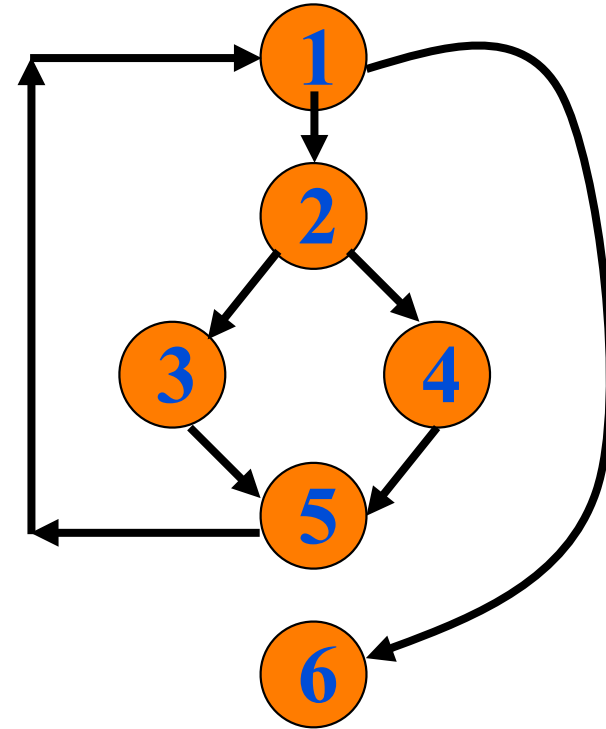   - E is the number of edges in G

Cyclomatic complexity

- The cyclomatic complexity of a method provides a lower bound on the number of test cases to be designed

  - to guarantee coverage of all linearly independent paths.

- Knowing the number of test cases required:

  - does not make it any easier to derive the test cases,
  - only gives an indication of the minimum number of test cases required.

hold

stop

Example

```
    int gcd(int x, int y) {
1.    while (x != y) {
2.        if (x>y)
3.            x = x-y;
4.        else y = y-x;
5.    }
6.    return x;
    }
```


- Number of independent paths = E-N+2 = 7-6+2 = **3**

- 1,6   test case (x=1, y=1)
- 1,2,3,5,1,6   test case (x=1, y=2)
- 1,2,4,5,1,6   test case (x=2, y=1)

E.g.
  {(x=0, y=0), (x=1, y=1)}
yields branch coverage. What
error might it miss?

```
if (x==0)
   f1();
else
   f2();

if (y==0)
   f3();
else
   f4();
return
```
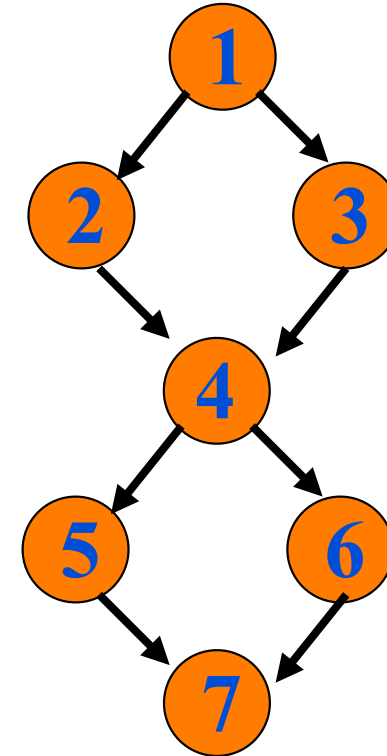
Applying Basis Path Coverage…

```
1. if (x==0)
2.    f1();
   else
3.    f2();

4. if (y==0)
5.    f3();
   else
6.    f4();
7. return
```


E = 8
N = 7
E-N+2 = **3**

Possible test cases for basis path coverage:
    (x=0, y=0),
    (x=0, y=1)
    (x=1, y=0)

Summary

- Exhaustive testing of non-trivial systems is impractical.

- Two possible approaches to testing are:

    - **black-box testing**: Code is hidden, based on specification only
    - **white-box testing**: based on the structure of the code

- White-box testing can involve: Statement coverage, branch coverage, condition coverage, path coverage.

- McCabe's Cyclomatic complexity metric:

    - provides an upper bound for number of linearly independent paths

hold