Faults, Errors, and Failures

For this course, we are going to define the following terminology.

- **Fault** (also known as a bug): A static defect in software—incorrect lines of code.
- **Error**: An incorrect internal state—not necessarily observed yet.
- **Failure**: External, incorrect behaviour with respect to the expected behaviour—must be visible (e.g. EPIC FAIL).

These terms are not used consistently in the literature. Don't get stuck on memorizing them.

Motivating Example. Here's a train-tracks analogy.

(all railroad pictures inspired by: Bernd Bruegge & Allen H. Dutoit, *Object Oriented Software Engineering: Using UML, Patterns and Java.*)

Is it a failure? An error? A fault? Clearly, it's not right, But no failure has occurred yet; there is no behaviour. I'd also say that nothing analogous to execution has occurred yet either. If there was a train on the tracks, pre-derailment, then there would be an error. That picture most closely corresponds to a fault.

Perhaps it was caused by mechanical stresses.

Or maybe it was caused by poor design.

Software-related Example. Let's get back to software and consider the code from last time:

```
public static numZero(int[] x) {
  // effects: if x is null, throw NullPointerException
  //          otherwise, return number of occurrences of 0 in x.
  int count = 0;
  for (int i = 1; i < x.length; i++) {
    // program point (*)
    if (x[i] == 0) count++;
  }
  return count;
}
```

As we saw, it has a fault (independent of whether it is executed or not): it's supposed to return the number of 0s, but it doesn't always do so. We define the state for this method to be the variables `x`, `i`, `count`, and the Program Counter (PC).

Feeding this `numZero` the input {`2, 7, 0`} shows a wrong state.

The **wrong state** is as follows: `x = {2, 7, 0}, i = 1, count = 0, PC = (*)`, on the first time around the loop.

The **expected state** is: `x = {2, 7, 0}, i = 0, count = 0, PC = (*)`

However, running `numZero` on {`2, 7, 0`} executes the fault and causes a (transient) error state, but doesn't result in a failure, as the output value `count` is 1 as expected.

On the other hand, running `numZero` on {`0, 2, 7`} causes an error state with `count = 0` on return, hence leading to a failure.

RIP Fault Model

To get from a fault to a failure:

1. Fault must be *reachable*;
2. Program state subsequent to reaching fault must be incorrect: *infection*; and
3. Infected state must *propagate* to output to cause a visible failure.

Applications of the RIP model: automatic generation of test data, mutation testing.

Dealing with Faults, Errors and Failures

Three strategies for dealing with faults are avoidance, detection and tolerance. Or, you can just try to declare that the fault is not a bug, if the specification is ambiguous.

Fault Avoidance. Certain faults can just be avoided by not programming in vulnerable languages; buffer overflows, for instance, are impossible in Java. Better system design can also help avoid faults, for instance by making an error state unreachable.

Fault Detection. Testing (construed broadly) is the primary means of fault detection. Software verification also qualifies. Once you have detected a fault, if it is economically viable, you might repair it:

Fault Tolerance. You are never going to remove all of the bugs, and some errors arise from conditions beyond your control (such as hardware faults). It's worthwhile to tolerate faults too. Strategies include redundancy and isolation. An example of redundancy is provisioning extra hardware in case a server goes down. Isolation includes things as simple as checking preconditions.

Testing vs Debugging

Recall from last time:

Testing: evaluating software by observing its execution.
Debugging: finding (and fixing) a fault given a failure.

I said that you really need to automate your tests. But even so, testing is still hard: only certain inputs expose the fault in the form of a failure. As you've experienced, debugging is hard too: you have the failure, but you have to find the fault.

Contrived example. Consider the following code:

```
if (x - 100 <= 0)
  if (y - 100 <= 0)
    if (x + y - 200 == 0)
      crash();
```

Only one input, `x = 100` and `y = 100`, will trigger the crash. If you're just going to do a random brute-force search over all 32-bit integers, you are never going to find the crash.

(We saw the `numZero` example today. That example is in the L02 lecture notes.)

Exercise: findLast. Here's a faulty program from last time, again.

```
static public int findLast(int[] x, int y) {
    /* bug: loop test should be i >= 0 */
    for (int i = x.length - 1; i > 0; i--) {
        if (x[i] == y) {
            return i;
        }
```

```
        }
        return -1;
    }
```

On the assignment, you will have a question like this:

> [(a)]Identify the fault, and fix it. If possible, identify a test case that does not execute the fault. If possible, identify a test case that executes the fault, but does not result in an error state. If possible, identify a test case that results in an error, but not a failure. (Hint: PC) For the given test case, identify the first error state. Be sure to describe the complete state.

I asked you to work on that question in class. (I noticed that most people were on-topic, thanks!) Here are some answers:

> [(a)]The loop condition must include index 0: `i >= 0`. This is a bit of a trick question. To avoid the loop condition, you must not enter the loop body. The only way to do that is to pass in `x = null`. You should also state, for instance, `y = 3`. The expected output is a `NullPointerException`, which is also the actual output. Inputs where `y` appears in the second or later position execute the fault but do not result in the error state; nor do inputs where `x` is an empty array. (There may be other inputs as well.) So, a concrete input: `x = {2, 3, 5}; y = 3`. Expected output = actual output = 1. One error input that does not lead to a failure is where `y` does not occur in `x`. That results in an incorrect PC after the final executed iteration of the loop. After running line 6 with `i = 1`, the decrement occurs, followed by the evaluation of `i > 0`, causing the PC to exit the loop (statement 8) instead of returning to statement 4. The faulty state is `x = {2, 3, 5}; y = 3; i = 0; PC = 8`, while correct state would be `PC = 4`.

Someone asked about distinguishing errors from failures. These questions were about failures at the method level, and so a wrong return value would be a failure when we're asking about methods. In the context of a bigger program, that return value might not be visible to the user, and so it might not constitute a failure, just an error.

Line Intersections

We then talked about different ways of validating a test suite. Consider the following Python code, found by Michael Thiessen on stackoverflow (http://stackoverflow.com/questions/306316/determine-if-two-rectangles-overlap-each-other).

```
class LineSegment:
    def __init__(self, x1, x2):
        self.x1 = x1; self.x2 = x2;

def intersect(a, b):
    return (a.x1 < b.x2) & (a.x2 > b.x1);
```

We could construct test suites that:

- execute every statement in `intersect` (node coverage, we'll see later). Well, that's not very useful; any old test case will do that. There are no branches, so what we'll call edge coverage doesn't help either.

- feed random inputs to `intersect`; unfortunately, interesting behaviours are not random, so it won't help much in general.
- check all outputs of `intersect` (i.e. a test case with lines that intersect and one with lines that don't intersect): we're getting somewhere—that will certify that the method works in some cases, but it's easy to think of situations that we missed.
- check different values of clauses `a.x1 < b.x2` and `a.x2 > b.x1` (logic coverage)—better than the above coverage criteria, but still misses interesting behaviours;
- analyze possible inputs and cover all interesting combinations of inputs (input space coverage)—can create an exhaustive test suite, if your analysis is sound.

Let's try to prove correctness of `intersect`. There's an old saying about testing—supposedly, it can only find the presence of bugs, not their absence. This is not completely true, especially if you have reliable software that automatically constructs an exhaustive test suite. But that is beyond the state of the practice in 2015, for the most part.

[Below is a cleaner presentation than the one in class. It also has the advantage of being not wrong.]

Inputs to `intersect`. There are essentially four inputs to this function. Rename them $aAbB$, for `a.x1`, `a.x2`, etc.

- Let's first assume that all points are distinct. We should make a note to ourselves to check violations of this, as well: we may have $a = A, a = b, a = B$, and symmetrically for B: $b = a, b = A, b = B$.
- For the purpose of the analysis, let's assume that $a < b$; when constructing testcases, we can swap a and b around. That's why there are duplicate assert statements below.
- Without loss of generality, we can assume that $a < A$ and $b < B$. (We ought to update the constructor if we want to make that assumption.)

With these assumptions, we have to test the three possible permutations $aAbB$, $abAB$, and $abBa$. It is simple to construct test cases for these permutations, using Python's unittest framework:

```
    def test_aAbB(self):
      a = LineSegment(0,2)
      b = LineSegment(3,7)
      self.assertFalse(intersect(a,b))
      self.assertFalse(intersect(b,a))

  def test_abAB(self):
      a = LineSegment(0,4)
      b = LineSegment(3,7)
      self.assertTrue(intersect(a,b))
      self.assertTrue(intersect(b,a))

  def test_abBA(self):
      a = LineSegment(0,4)
      b = LineSegment(1,2)
      self.assertTrue(intersect(a,b))
      self.assertTrue(intersect(b,a))
```

Those test cases pass. However, if you construct test cases for equality (as I've committed to the repository), you see that the given `intersect` function fails on line segments that intersect only at a point. Replacing `<` with `<=` and `>` with `>=` fixes the code.

About Testing

We can look at testing statically or dynamically.

Static Testing (ahead-of-time): this includes static analysis, which is typically automated and runs at compile time (or, say, nightly), as well human-driven static testing—walk-throughs (informal) and code inspection (formal).

Dynamic Testing (at run-time): observe program behaviour by executing it; includes black-box testing and white-box testing.

Usually the word "testing" means *dynamic testing*.

Naughty words. People like to talk about "complete testing", "exhaustive testing", and "full coverage". However, for many systems, the number of potential inputs is infinite. It's therefore impossible to completely test a nontrivial system, i.e. run it on all possible inputs. There are both practical limitations (time and cost) and theoretical limitations (i.e. the halting problem).

In the absence of complete testing, we will define *testing criteria* and evaluate test suites with them.

Test cases

Informally, a *test case* contains:

- what you feed to software; and
- what the software should output in response.

Here are two definitions to help evaluate how hard it might be to create test cases.

Definition 1 Observability *is how easy it is to observe the system's behaviour, e.g. its outputs, effects on the environment, hardware and software.*

Definition 2 Controlability *is how easy it is to provide the system with needed inputs and to get the system into the right state.*

Anatomy of a Test Case

Consider testing a cellphone from the "off" state:

$$\underbrace{\langle \text{ on } \rangle}_{\text{prefix values}} \quad \underbrace{1\ 519\ 888\ 4567}_{\text{test case values}} \quad \overbrace{\underbrace{\langle \text{ talk } \rangle}_{\text{verification values}} \quad \underbrace{\langle \text{ end } \rangle}_{\text{exit codes}}}^{\text{postfix values}}$$

Definition 3

- Test Case Values*: input values necessary to complete some execution of the software. (often called the test case itself)*
- Expected Results*: result to be produced iff program satisfies intended behaviour on a test case.*
- Prefix Values*: inputs to prepare software for test case values.*
- Postfix Values*: inputs for software after test case values;*
  - verification values*: inputs to show results of test case values;*
  - exit commands*: inputs to terminate program or to return it to initial state.*

Definition 4

- Test Case*: test case values, expected results, prefix values, and postfix values necessary to evaluate software under test.*
- Test Set*: set of test cases.*
- Executable Test Script*: test case prepared in a form to be executable automatically and which generates a report.*

On Coverage

Ideally, we'd run the program on the whole input space and find bugs. Unfortunately, such a plan is usually infeasible: there are too many potential inputs.

Key Idea: Coverage. Find a reduced space and cover that space.

We hope that covering the reduced space is going to be more exhaustive than arbitrarily creating test cases. It at least tells us when we can plausibly stop testing.

The following definition helps us evaluate coverage.

Definition 5 *A* test requirement *is a specific element of a (software) artifact that a test case must satisfy or cover.*

We write TR for a set of test requirements; a test set may cover a set of TRs.

For instance, consider three ice cream cone flavours: vanilla, chocolate and mint. A possible test requirement would be to test one chocolate cone. (Volunteers?)

Two software examples:

- cover all decisions in a program (branch coverage); each decision gives two test requirements: branch is true; branch is false.
- each method must be called at least once; each method gives one test requirement.

Definition 6 *A* coverage criterion *is a rule or collection of rules that impose test requirements on a test set.*

A test set may or may not satisfy a coverage criterion. The coverage criterion gives a recipe for generating TRs systemically.

Returning to the ice cream example, a flavour criterion might be "cover all flavours", and that would generate three TRs: {flavour: chocolate, flavour: vanilla, flavour: mint}.

We can test an ice cream stand by running two test sets on it, for instance: test set 1 includes 3 chocolate cones and 1 vanilla cone, while test set 2 includes 1 chocolate cone, 1 vanilla cone, and 1 mint cone.

Definition 7 *(Coverage). Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C iff for every test requirement $tr \in TR$, at least one $t \in T$ satisfies TR.*

Infeasible Test Requirements. Sometimes, no test case will satisfy a test requirement. For instance, dead code can make statement coverage infeasible, e.g.:

```
if (false)
  unreachableCall();
```

or, a real example from the Linux kernel:

```
while (0)
  {local_irq_disable();}
```

Hence, a criterion which says "test every statement" is going to be infeasible for many programs.

Quantifying Coverage. How good is a test set? It's great if it covers everything, but sometimes that's impossible. We can instead assign a number.

Definition 8 *(Coverage Level). Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.*

Returning to our example, say TR = {flavour: chocolate, flavour: vanilla, flavour: mint}, and test set T1 contains {3 chocolate, 1 vanilla}, then the coverage level is "2/3" or about 67%.

Subsumption

Sometimes one coverage criterion is strictly more powerful than another one: any test set that satisfies C_1 might automatically satisfy C_2.

Definition 9 *Criteria subsumption: coverage criterion* C_1 subsumes C_2 *iff* every *test set that satisfies* C_1 *also satisfies* C_2*.*

Software example: branch coverage ("Edge Coverage") subsumes statement coverage ("Node Coverage"). Which is stronger?

Evaluating coverage criteria. Subsumption is a rough guide for comparing criteria, but it's hard to use in practice. Consider also:

1. difficulty of generating test requirements;
2. difficulty of generating tests;
3. how well tests reveal faults.