**Software Testing, Quality Assurance and Maintenance** Winter 2015

# Lecture 2 — January 7, 2015

*Patrick Lam* *version 1*

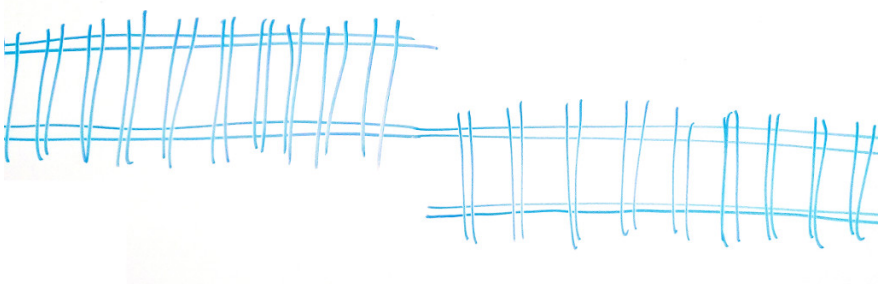
## Faults, Errors, and Failures

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.

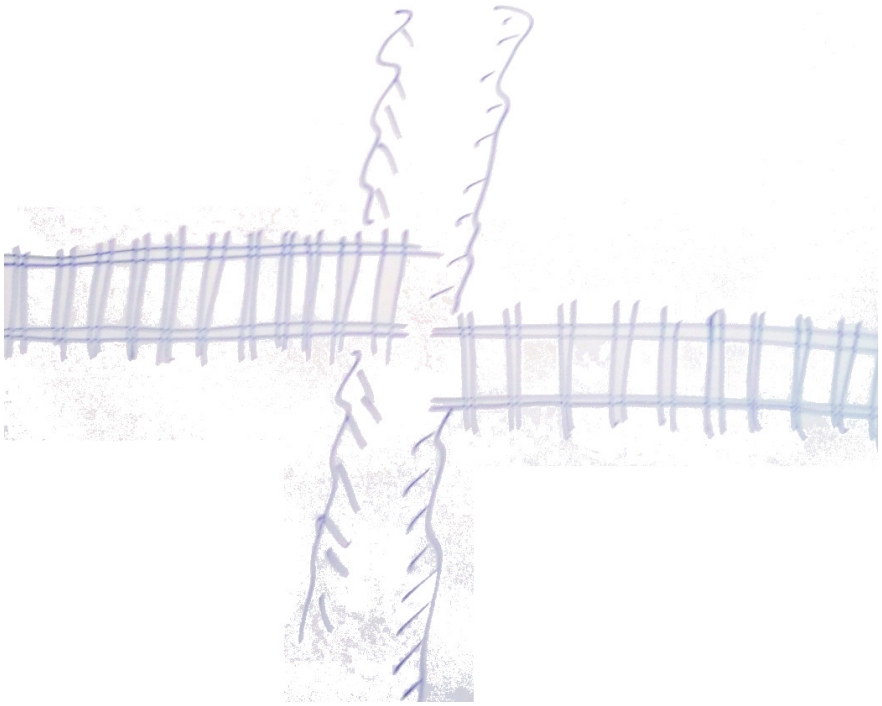
**Motivating Example.** Here's a train-tracks analogy.



(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 this code:

```
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.

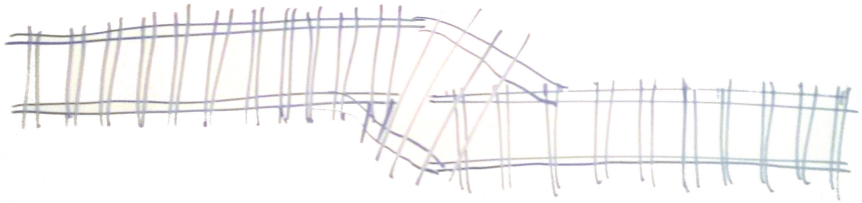
Applications of the RIP model: automatic generation of test data, mutation testing.

## Dealing with Faults, Errors and Failures

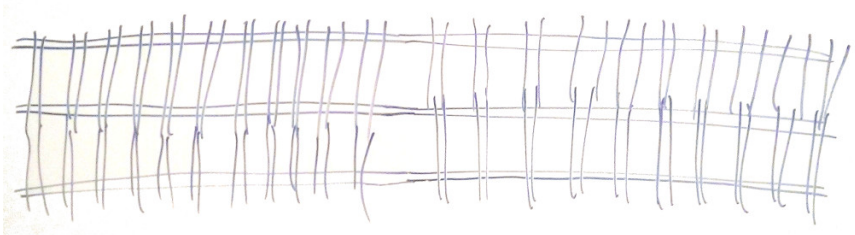
Three strategies for dealing with faults are avoidance, detection and tolerance. Or, you can just try to declare that the fault is not a bug, if the specification is ambiguous.

**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.