Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 2 — January 7/9, 2017

Patrick Lam *version 1*

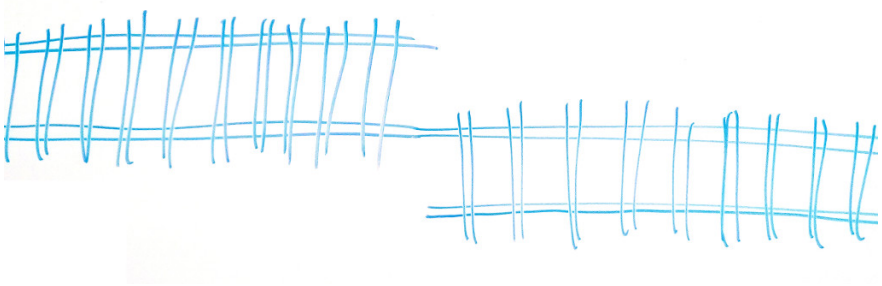
Faults, Errors, and Failures

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.

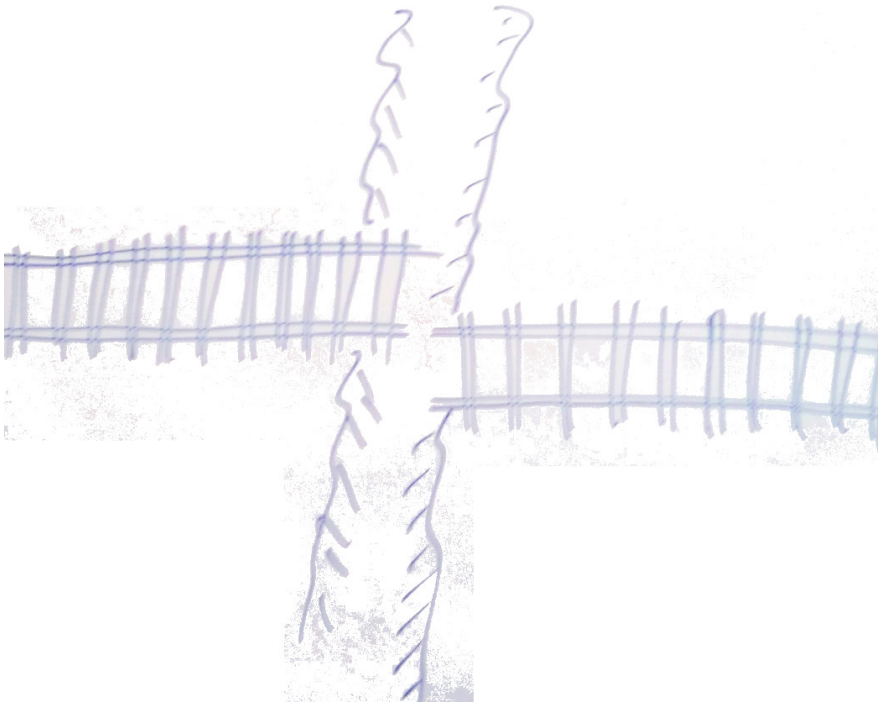
Motivating Example. Here's a train-tracks analogy.



(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.

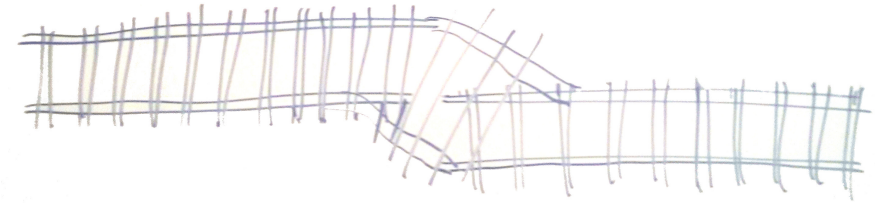
Applications of the RIP model: automatic generation of test data, mutation testing.

Dealing with Faults, Errors and Failures

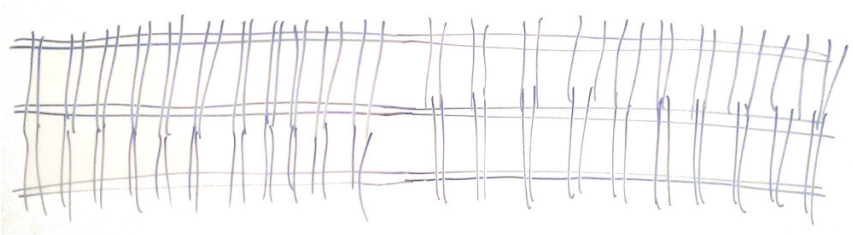
Three strategies for dealing with faults are avoidance, detection and tolerance. Or, you can just try to declare that the fault is not a bug, if the specification is ambiguous.

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.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 3 — January 7/9, 2019

Patrick Lam *version 0*

Exercise: findLast. Here's a faulty program.

```
1     static public int findLast(int[] x, int y) {
2         /* bug: loop test should be i >= 0 */
3         for (int i = x.length - 1; i > 0; i--) {
4             if (x[i] == y) {
5                 return i;
6             }
7         }
8         return -1;
9     }
```

You might expect to be asked a question like this:

(a) Identify the fault, and fix it.

(b) If possible, identify a test case that does not execute the fault.

(c) If possible, identify a test case that executes the fault, but does not result in an error state.

(d) If possible, identify a test case that results in an error, but not a failure. (Hint: program counter)

(e) 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. Here are some answers:

(a) The loop condition must include index 0: `i >= 0`.

(b) 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.

(c) 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.

(d) 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.

(e) 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`).

```
1 class LineSegment:
2     def __init__(self, x1, x2):
3         self.x1 = x1; self.x2 = x2;
4
5 def intersect(a, b):
6     return (a.x1 < b.x2) & (a.x2 > b.x1);
```

We could construct test suites that:

- execute every statement in `intersect` (statement coverage). 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.

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 test cases, 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:

```
1       def test_aAbB(self):
2         a = LineSegment(0,2)
3         b = LineSegment(3,7)
4         self.assertFalse(intersect(a,b))
5         self.assertFalse(intersect(b,a))
6
7     def test_abAB(self):
8         a = LineSegment(0,4)
9         b = LineSegment(3,7)
10        self.assertTrue(intersect(a,b))
11        self.assertTrue(intersect(b,a))
12
13    def test_abBA(self):
14        a = LineSegment(0,4)
15        b = LineSegment(1,2)
16        self.assertTrue(intersect(a,b))
17        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.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 4 — January 14, 2019

Patrick Lam *version 1*

Last time. We saw two examples: `findLast`, where we did an example about identifying faults, errors, and failures, and finding distinguishing test cases; and the line intersection example, where we informally reasoned about the correctness of that algorithm and demonstrated it using testing. The reasoning is a lot more ad-hoc than what we've seen in algorithms courses.

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—typically code review.

Dynamic Testing (at run-time): observe program behaviour by executing it; includes black-box testing (not looking at code) and white-box testing (looking at code to develop tests).

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).

The first part of the course is about defining test suites and aims to give you tools to answer the question "when should I stop testing?" Reiterating the syllabus, I might stop:

- *When I run out of time.* Open-ended exploratory testing; automatic input generation.
- *When I am close enough to being exhaustive.* Coverage of: (enough) statements, branches, program states, use cases.

I'll again put in a shout-out to mutation as being another way to validate whether one's test suite is good enough.

Test cases

As we've seen in the last two lectures, a *test case* contains:

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

Our test cases have been easy to generate so far, but that's not always the case.

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.

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.

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 6 *(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.*

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 5 — January 14/16, 2019

Patrick Lam *version 2*

Last time. We spent most of the time talking about the distinction between static and dynamic analysis of code. Static analysis finds all the things (perhaps too many), while dynamic analysis tells you exactly what happens on a particular execution (but you have to have the right inputs). We also looked at a JUnit test case (setup/teardown/establishing state/checking system under test actions); the test case we looked at was at `https://github.com/google/guava/blob/master/guava-tests/test/com/google/common/io/MoreFilesTest.java`. Finally, we introduced the notion of coverage and test requirements a bit more formally.

Exploratory Testing

Exploratory testing is usually (but not always) carried out by dedicated testers. In that sense, it's somewhat different from the other testing activities in this course, which are more developer-focussed—our usual goal is learning, as developers, how to deploy better automated test suites for our software. Hallway usability testing, though, is an application of exploratory testing. Furthermore, the dedicated QA function is important, and we should learn about how it works.

Resources. James Bach has a shorter and a longer introduction to exploratory testing:

- `http://www.satisfice.com/articles/what_is_et.shtml`
- `http://www.satisfice.com/articles/et-article.pdf`

There is an exhaustive set of notes on exploratory testing by Cem Kaner:

- `http://www.kaner.com/pdfs/QAIExploring.pdf`

> "Exploratory testing is simultaneous learning, test design, and test execution."

Contrast this to scripted testing: test design happens ahead of time and then test execution happens (repeatedly) throughout the product's development cycle. When we think of dedicated QA teams, we think they are manually executing scripted tests. In 2019, that is not an effective use of staff.

There is a continuum between scripted testing and exploratory testing. Good exploratory testing may use prepared scripts for certain tasks.

Scenarios where Exploratory Testing Excels. (from Bach's article)

- providing rapid feedback on new product/feature;
- learning product quickly;
- diversifying testing beyond scripts;
- finding single most important bug in shortest time;
- independent investigation of another tester's work;
- investigating and isolating a particular defect;
- investigate status of a particular risk to evaluate need for scripted tests.

Exploratory Testing Process. Exploratory testing should not be randomly bumbling around (we can call that "ad hoc testing")—the random approach finds bugs but isn't the most efficient at giving you an idea of how well the software works.

- Start with a charter for your testing activity, e.g. "Explore and analyze the product elements of the software." These charters should be somewhat ambiguous.
- Decide what area of the software to test.
- Design a test (informally).
- Execute the test; log bugs.
- Repeat.

Exploratory testing shouldn't produce an exhaustive set of notes. Good testers will be able to reproduce the bugs that they encounter during their testing from brief notes. Taking full notes takes too long.

The output from exploratory testing is at least a set of bug reports. It may also include test notes, which include overall impressions and a summary of the test strategy/thought process. Artifacts such as test data or test materials are also both inputs and outputs from exploratory testing.

Primary vs contributing tasks. One way to classify tasks that software can do (or, in other words, its features) is *primary* vs *contributing*. A *primary* task is core functionality of the system; it's something that you would say "You Had One Job!" about. As examples, text editors must be able to load text files, add text, and save the text files. On the other hand, *contributing* tasks are secondary. A macro system for a text editor would be a contributing task. Being able to read email in your editor is definitely a contributing task. Sometimes it's not black-and-white. Spell-check can go either way.

Example. I recommend reading the example by James Bach in "et-article.pdf" about the photo editing program ("ET in Action"). He describes how he used ET to evaluate software for Windows compatibility and found critical defects.

In-class exercise: Exploratory testing of WaterlooWorks.

We will try out exploratory testing with WaterlooWorks. I believe that everyone should have access to the system, although for some of you there may be no jobs visible right now.

The charter will be "Explore the overall functionality of WaterlooWorks". Summarize in one or two sentences what the purpose of WaterlooWorks is. Identify the tasks that WaterlooWorks should be able to do and classify them as primary or contributing. Identify areas of potential instability. Test each function and record results (bugs).

Of course, don't do things that have actual effects. Usually, testers would have access to a development server and could test those areas more aggressively. But we are working with production systems here.

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 6 — January 16, 2019

Patrick Lam *version 1*

Last time. We discussed benefits of exploratory testing and a sketch of how to do it. We also started to do a case study of WaterlooWorks.

Source Code Coverage Criteria

We alluded to statement coverage and branch coverage. It is possible to evaluate these criteria directly on the source code, but better to use a sensible intermediate representation. The fundamental graph for source code is the *Control-Flow Graph* (CFG), which originates from compilers.

- CFG nodes: a node represents zero or more statements;
- CFG edges: an edge (s_1, s_2) indicates that s_1 may be followed by s_2 in an execution.

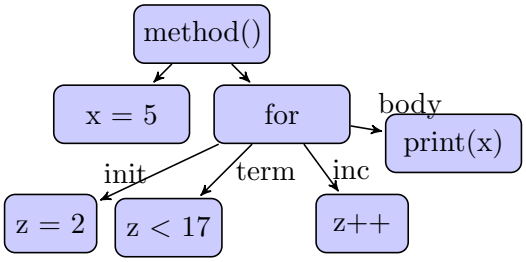
Example. Consider the following code.

```
1   x = 5;
2   for (z = 2; z < 17; z++)
3     print(x);
```

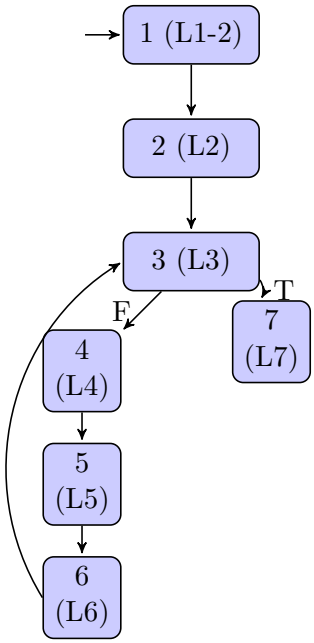
Recall the steps in compilation:

- lexing: input = stream of characters, output = stream of tokens (if, while, strings)
- parsing: input = stream of tokens, output = concrete syntax tree
- construction of Abstract Syntax Tree (AST): cleans up the concrete syntax tree
- conversion to Control Flow Graph: input = AST, output = CFG
- optimizations: input = CFG, output = CFG
- convert to bytecode/machine code: input = CFG, output = bytecode/machine code

The Abstract Syntax Tree corresponding to the example code might look like this:



From ASTs to CFGs. We can convert the Abstract Syntax Tree into the following Control Flow Graph (and we'll see how to do so in Lecture 7).



From CFG to low-level code. And we can convert the CFG into the following low-level code:

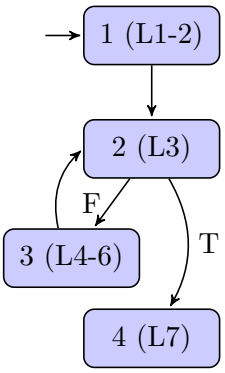
```
1       x = 5
2       z = 2
3   q0: if (z < 17) goto q1
4       z = z + 1
5       print (x)
6       goto q0
7   q1: nop
```

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 7 — January 21, 2019

Patrick Lam *version 1*

Basic Blocks. We can simplify a CFG by grouping together statements which always execute together (in sequential programs):



We use the following definition:

Definition 1 *A basic block is a sequence of instructions in the control-flow graph that has one entry point and one exit point.*

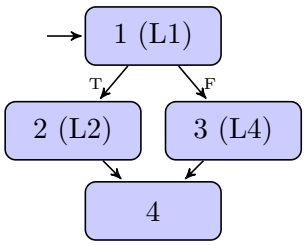
We are usually interested in forming maximal basic blocks. Note that a basic block may have multiple successors. However, there may not be any jumps into the middle of a basic block (which is why statement `l0` has its own basic block.)

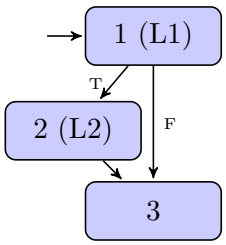
Some Examples

We'll now see how to construct control-flow graph fragments for various program constructs.

if statements: One can put the conditions (and hence uses) on the control-flow edges, rather than in the `if` node. I prefer putting the condition in the node.

```
1  if (z < 17)
2    print (x);
3  else
4    print (y);
```



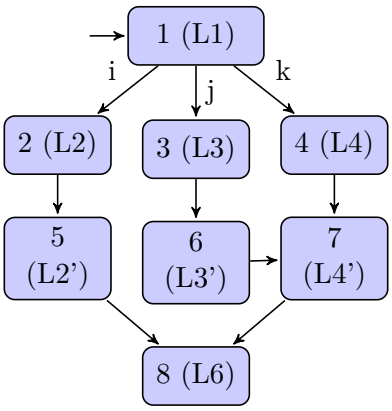



```
1  if (z < 17)
2    print(x);
```

Short-circuit `if` evaluation is more complicated; I recommend working it out yourself.

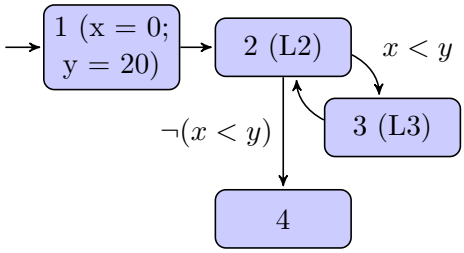
case / switch statements:

```
1  switch (n) {
2    case `I': ...; break;
3    case `J': ...; // fall thru
4    case `K': ...; break;
5  }
6  // ...
```



while statements:

```
1  x = 0; y = 20;
2  while (x < y) {
3    x ++; y --;
4  }
```



Note that arbitrarily complicated structures may occur inside the loop body.

for statements:

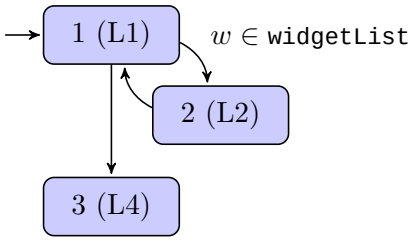
```
1  for (int i = 0; i < 57; i++) {
2    if (i % 3 == 0) {
3      print (i);
4    }
5  }
```

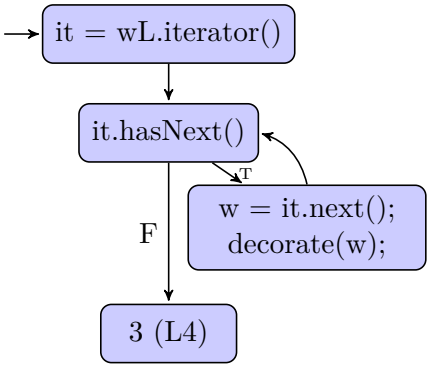
(an exercise for the reader;
we saw one earlier!)

This example uses Java's enhanced for loops, which iterates over all of the elements in the `widgetList`:

```
1  for (Widget w : widgetList) {
2    decorate(w);
3  }
```

I will accept the simplified CFG or the more useful one on the right:





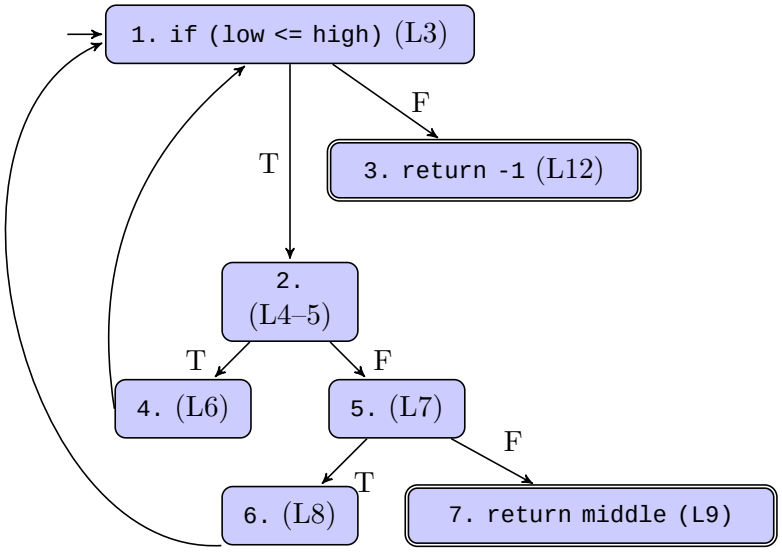
Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 8 — January 23, 2019

Patrick Lam *version 1*

Larger CFG example. You can draw a 7-node CFG for this program:

```
1   /** Binary search for target in sorted subarray a[low..high] */
2   int binary_search(int[] a, int low, int high, int target) {
3     while (low <= high) {
4       int middle = low + (high-low)/2;
5       if (target < a[middle)
6         high = middle - 1;
7       else if (target > a[middle])
8         low = middle + 1;
9       else
10        return middle;
11    }
12    return -1; /* not found in a[low..high] */
13  }
```



Here are more exercise programs that you can draw CFGs for.

```
1  /* effects: if x==null, throw NullPointerException
2              otherwise, return number of elements in x that are odd, positive or both. */
3  int oddOrPos(int[] x) {
4    int count = 0;
5    for (int i = 0; i < x.length; i++) {
6      if (x[i]%2 == 1 || x[i] > 0) {
7        count++;
8      }
9    }
10   return count;
11 }
12
13 // example test case: input: x=[-3, -2, 0, 1, 4]; output: 3
```

Finally, we have a really poorly-designed API (I'd give it a D at most, maybe an F) because it's impossible to succinctly describe what it does. **Do not design functions with interfaces like this.** But we can still draw a CFG, no matter how bad the code is.

```
1   /** Returns the mean of the first maxSize numbers in the array,
2       if they are between min and max. Otherwise, skip the numbers. */
3   double computeMean(int[] value, int maxSize, int min, int max) {
4     int i, ti, tv, sum;
5
6     i = 0; ti = 0; tv = 0; sum = 0;
7     while (ti < maxSize) {
8       ti++;
9       if (value[i] >= min && value[i] <= max) {
10        tv++;
11        sum += value[i];
12      }
13      i++;
14    }
15    if (tv > 0)
16      return (double)sum/tv;
17    else
18      throw new IllegalArgumentException();
19  }
```

Statement and Branch Coverage

We defined Control-Flow Graphs so that we can give principled definitions of statement and branch coverage. We can start with the definition of a test path:

Definition 1 *A* test path *is a path p (possibly of length 0) that starts at some initial node (i.e. in N_0) and ends at some final node (i.e. in N_f).*

Here's a definition of coverage for graphs:

Definition 2 *Given a set of test requirements* TR *for a graph criterion C, a test set T satisfies C on graph G iff for every test requirement* tr *in* TR*, at least one test path p in path(T) exists such that p satisfies* tr*.*

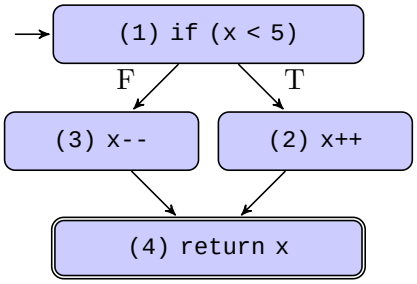
We'll use this notion to define a number of standard testing coverage criteria. But first, what are test paths?

Test cases and test paths. We connect test cases and test paths with a mapping path_G from test cases to test paths; e.g. $\text{path}_G(t)$ is the set of test paths corresponding to test case t.

- usually we just write path since G is obvious from the context.
- we can lift the definition of path to test sets T by defining $\text{path}(T) = \{\text{path}(t) | t \in T\}$.
- each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

Example. Here is a short method, the associated control-flow graph, and some test cases and test paths.

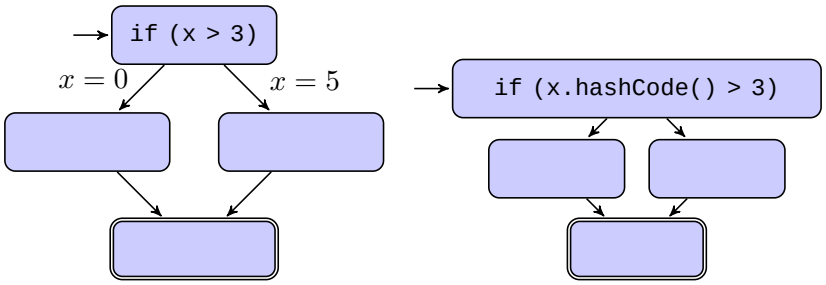
```
1  int foo(int x) {
2    if (x < 5) {
3      x ++;
4    } else {
5      x --;
6    }
7    return x;
8  }
```



- Test case: $x = 5$; test path: $[(1), (3), (4)]$.
- Test case: $x = 2$; test path: $[(1), (2), (4)]$.

Note that (1) we can deduce properties of the test case from the test path; and (2) in this example, since our method is deterministic, the test case determines the test path.

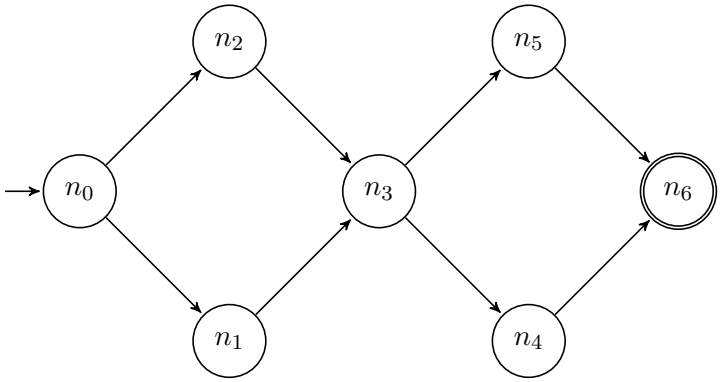
Nondeterminism. I mentioned the mapping between test cases and test paths above. The mapping is not one-to-one for nondeterminstic code. Here's an example of deterministic and nondeterministic control-flow graphs:



Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.

As another (more abstract) example, consider the double-diamond graph D.



Here are the four test paths in D:

$$[n_0, n_1, n_3, n_4, n_6]$$
$$[n_0, n_1, n_3, n_5, n_6]$$
$$[n_0, n_2, n_3, n_4, n_6]$$
$$[n_0, n_2, n_3, n_5, n_6]$$

For the *statement coverage* criterion, we get the following test requirements:

$$\{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}$$

That is, any test set T which satisfies statement coverage on D must include test cases t; the cases t give rise to test paths path(t), and some path must include each node from n_0 to n_6. (No single path must include all of these nodes; the requirement applies to the set of test paths.)

Let's formally define statement coverage.

Definition 3 *Statement coverage: For each node $n \in reach_G(N_0)$,* TR *contains a requirement to visit node n.*

For our example,

$$TR = \{n_0, n_1, n_2, n_3, n_4, n_5, n_6\}.$$

Let's consider an example of a test set which satisfies statement coverage on D.

Start with a test case t_1; assume that executing t_1 gives the test path

$$\text{path}(t_1) = p_1 = [n_0, n_1, n_3, n_4, n_6].$$

Then test set $\{t_1\}$ does not give statement coverage on D, because no test case covers node n_2 or n_5. If we can find a test case t_2 with test path

$$\text{path}(t_2) = p_2 = [n_0, n_2, n_3, n_5, n_6],$$

then the test set $T = \{t_1, t_2\}$ satisfies statement coverage on D.

What is another test set which satisfies statement coverage on D?

Here is a more verbose definition of statement coverage.

Definition 4 *Test set T satisfies* statement coverage *on graph G if and only if for every syntactically reachable node $n \in N$, there is some path p in path(T) such that p visits n.*

A second standard criterion is that of branch coverage.

Criterion 1 Branch Coverage. *TR contains each reachable path of length up to 1, inclusive, in G.*

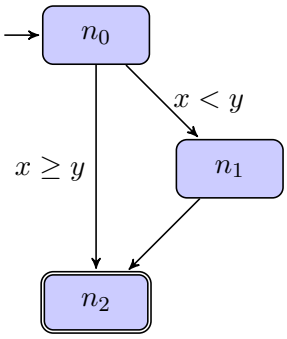
Here are some examples of paths of length ≤ 1:

Note that since we're not talking about *test paths*, these reachable paths need not start in N_0.

In general, paths of length ≤ 1 consist of nodes and edges. (Why not just say edges?)

Saying "edges" on the above graph would not be the same as saying "paths of length ≤ 1".

Another example. Here is a more involved example:



Let's define

$$\begin{aligned} \text{path}(t_1) &= [n_0, n_1, n_2] \\ \text{path}(t_2) &= [n_0, n_2] \end{aligned}$$

Then

T_1	=	$\langle ? \rangle$	satisfies statement coverage
T_2	=	$\langle ? \rangle$	satisfies branch coverage

Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 9 — January 23, 2019

Patrick Lam *version 1*

Selenium[1]

Selenium is a tool for (among other things) testing web applications. In its full generality, "Selenium automates browsers" and lets you programmatically drive Web browsers. For the purposes of this class, you can use Selenium to automate web application tests, testing multiple browsers and multiple platforms.

In Assignment 1, I asked you to use Selenium to test a web application. I've simplified the setup by not actually launching a browser; Selenium contains a "headless" (GUI-less) browser simulator called `HtmlUnit`. Selenium can also drive real browsers, and it's interesting to watch it do that. Furthermore, you can set up a bunch of different computers and drive them from a single Selenium script.

Selenium also includes record-replay functionality (the Selenium IDE, a Firefox add-on), but I won't discuss it.

Anatomy of a Selenium Test. There are many bindings for Selenium, but we'll talk about using Selenium from JUnit.

Like any other test, there is setup and teardown; for Selenium, you ask for the browser to be started and terminated.

```
1   @Before
2   public void openBrowser() {
3     baseUrl = System.getProperty("webdriver.base.url");
4     driver = new FirefoxDriver(); // could also be Chrome, IE, HtmlUnit, etc.
5     driver.get(baseUrl);
6   }
7
8   @After
9   public void saveScreenshotAndCloseBrowser() throws IOException {
10    driver.quit();
11  }
```

Note that we create a browser and ask it to load a page.

[1] `http://seleniumhq.org`

Now let's look at a particular test. Like any other test, it sets up the test state, calls upon the system under test to do the action, and checks the result.

```
1   @Test
2   public void testPageTitleAfterSearchShouldBeginWithDrupal() throws IOException {
3     assertEquals("The page title should equal Google at the start of the test.",
4                  "Google", driver.getTitle());  // (1)
5     WebElement searchField = driver.findElement(By.name("q")); // (2)
6     searchField.sendKeys("Drupal!"); // (3)
7     searchField.submit();            // (3)
8     assertTrue("The page title should start with the search string after the search.",
          // (4)
9             (new WebDriverWait(driver, 10)).until(new ExpectedCondition<Boolean>() {
10                    public Boolean apply(Object d) {
11                        return ((WebDriver)d).getTitle().toLowerCase().startsWith("
                              drupal!");
12                    }
13                })
14            );
15  }
```

Here's what's going on.

1. The initial `assertEquals` is based on an assumption that the test is called on `www.google.com`.
2. Next, the test searches for the appropriate input on the webpage. (For Java tests, we don't need to search for the method that we're calling. Here, we do.) Selenium includes various ways to search the Document Object Model.
3. Having found the appropriate field, the test sends input and submits the form. This is analogous to calling the methods on the System Under Test.
4. Finally, the test waits for the action to complete (with `WebDriverWait`). Once complete, it checks that the outputs are correct; in this case, it checks that the title of the resulting page starts with "drupal!".

Waiting. In the above example, the test simply waits until the page loads (or 10 seconds). Selenium tests can also wait for more sophisticated conditions, for instance until a certain element appears on the page. (This is relevant when the JavaScript is adding content to the page dynamically).

Example:

```
1 WebDriverWait wait = new WebDriverWait(driver, 10);
2 WebElement element =
3   wait.until(ExpectedConditions.elementToBeClickable(By.id("someid")));
```

Page Objects

The above example hard-coded the "q" field on Google's page. But things change. We can fix that by introducing a level of indirection.

References:

`http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#chapter06-reference`
`https://martinfowler.com/bliki/PageObject.html`

A page object should abstractly represent the actions that a user can take on a page and allow the caller to query the state of the page (if necessary). The Selenium documentation suggests a page object for a sign-in page:

```
1  public class SignInPage {
2    private WebDriver driver;
3
4    public SignInPage(WebDriver driver) {
5      this.driver = driver;
6      if(!driver.getTitle().equals("Sign in page")) {
7        throw new IllegalStateException("This is not sign in page, current page is: "
8        +driver.getLocation());
9      }
10   }
11
12   public HomePage loginValidUser(String userName, String password) {
13     driver.type("usernamefield", userName);
14     driver.type("passwordfield", password);
15     driver.click("sign-in");
16     driver.waitForPageToLoad("waitPeriod");
17
18     return new HomePage(driver);
19   }
20 }
```

The documentation suggests that the only verification that should occur on a page object is testing that the driver points to the right page. Everything else should be done in the tests.

The sign-in page object also exposes the sole functionality of the page, which is to sign in. It then returns the page that gets loaded after sign-in.

Note that the caller does not need to know about the identity of the username and password fields. In fact, let's say that a new version of the app included Facebook OAuth login. It would suffice to change the `SignInPage` to use the new login functionality; no other parts of the test suite would need to change.

In assignment 2, I will ask you to implement a fairly low-level Page Object, which simply passes the field contents directly to the caller. The sign-in example demonstrates a higher level of abstraction.