SE 465 Midterm Review

Faults, Errors & Failures

- **Validation** – evaluating software prior to release to ensure compliance with intended usage
- **Verification** – whether products of a given phase of the development process fulfill requirements established in a previous phase

- **Fault (bug)** – static defect in software; e.g. incorrect lines of code
- **Error** – an incorrect internal state that is the manifestation of some fault; not necessarily observed yet
- **Failure** – external, incorrect behaviour with respect to the expected behaviour; must be visible
- RIP fault model – for a fault to become a failure:
  - Fault must be reachable
  - Program state after reaching fault must be incorrect (i.e. infection)
  - Infected state must propagate to output to cause a visible failure
- Dealing with faults:
  - **Fault avoidance** – not programming in a vulnerable language; better system design, e.g. by making an error state unreachable
  - **Fault detection** – testing; software verification; and repairing detected faults
  - **Fault tolerance** – redundancy (e.g. extra hardware); isolation (e.g. checking preconditions)

- **Testing** – evaluating software by observing its execution
- **Debugging** – finding (and fixing) a fault given a failure

- Example:

```
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      }
```

  - A test case that does not execute the fault

    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.

  - A test case that executes the fault, but does not result in an error state

    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.

  - A test case that results in an error, but not a failure

    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.

- Identify the first error state

  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`.

Testing

- To test a program, we can:
  - Execute every statement in the program (**statement coverage**)
  - Feed random inputs
  - Check different values for output conditions (**logic coverage**)
  - Analyze possible inputs & cover all interesting combinations of input (**input space coverage**)
- Example:

```
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);
```

  - Rename points to a, A, b, B
  - Without loss of generality, assume a < A, b < B, and a < b
  - Then there are only 3 logical cases: aAbB, abAB, abBA

- **Static testing** – a.k.a. "ahead of time"
  - Static analysis – runs at compile time
  - Code review
- **Dynamic testing** – a.k.a. "at run-time"
  - Observe program behaviour by executing it
  - **Black-box** (not looking at code) & **white-box** testing (looking at code)
- When to stop testing?
  - When I run out of time
    - E.g. exploratory testing; automated input generation
  - When I am close enough to being exhaustive
    - Statement/branch/program state/use case coverage

- **Observability** – how easy it is to observe the system's behaviour; e.g. its outputs, effects on the environment, hardware and software
- **Controllability** – how easy it is to provide the system with needed inputs and to get the system into the right state

Test Cases & Coverage

- A **test case** consists of:
  - Test case values: input values necessary to complete some execution of the software
  - Expected results: result to be produced if & only if program satisfies intended behaviou
  - 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
- **Test set** – set of test cases

- **Coverage** – find a reduced input space and cover that space with tests
- **Test requirement (TR)** – a specific element of a (software) artifact that a test case must satisfy or cover
  - E.g. to achieve branch coverage, each branch gives 2 TRs (branch is true; branch is false)
  - Infeasible test requirements – e.g. dead code
- **Coverage level** = # of TRs satisfied by test set T/size(TR)

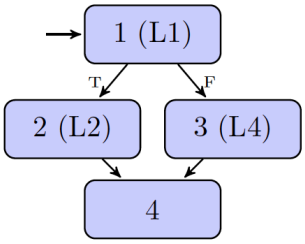
Exploratory Testing

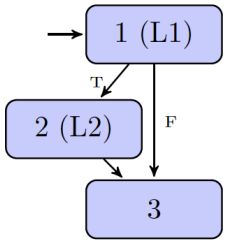
- **Exploratory testing** – "simultaneous learning, test design, and test execution"
  - In contrast: scripted testing – test design happens ahead of time, then execution occurs repeatedly
  - Process (not ad hoc testing):
    - Start with a charter for your testing activity, e.g. "Explore and analyze 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 (take notes – shouldn't be exhaustive)
    - Repeat
  - Scenarios where exploratory testing excels:
    - 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.

Coverage Criteria & CFGs

- **Control-flow graphs**
  - Node = zero or more statements (of code)
  - Edge = (s1, s2) → s1 may be followed by s2 in execution
- Examples:
  - Conditionals:

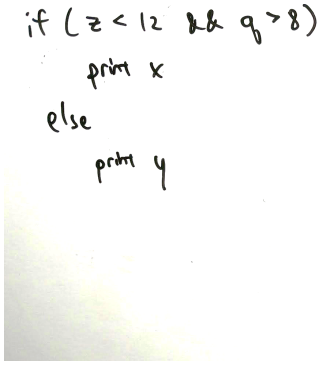
```
1  if (z < 17)
2    print (x);
3  else
4    print (y);
```

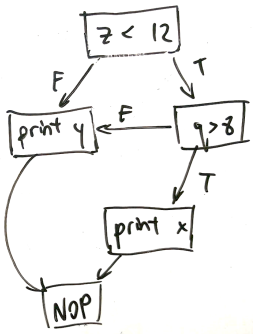





```
1  if (z < 17)
2    print(x);
```

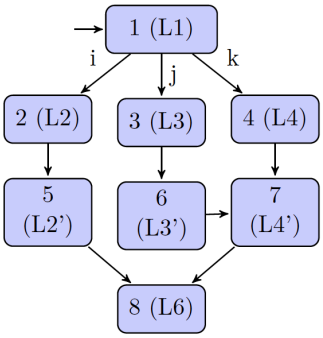
  - Short-circuit conditional:





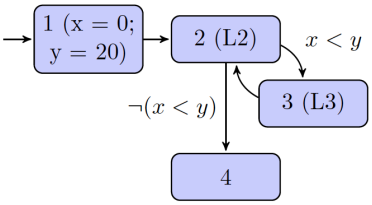
  - Switch statements:

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



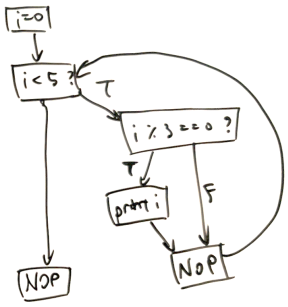
  - While loop:

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

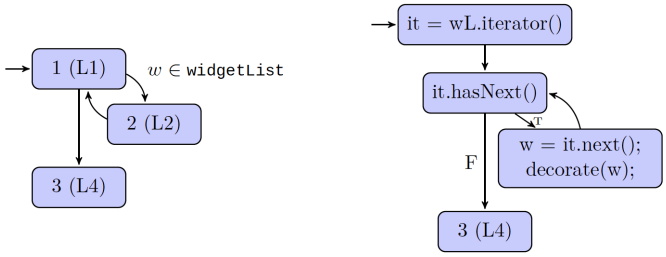


- For loop:

```
for (i=0 ; i<5 ; i++)
    if (i%3==0)
        print i
```

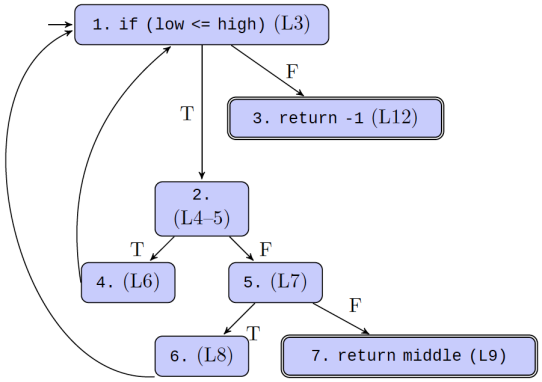


- Iterator for loop:



- Complex example:

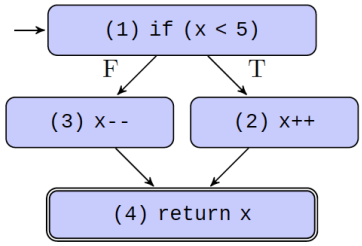
```
1   /** Binary search for target in sorted subarray a[low..high] */
2   int binary_search(int[] a, int low, int high, int target) {
3     while (low <= high) {
4       int middle = low + (high-low)/2;
5       if (target < a[middle)
6         high = middle - 1;
7       else if (target > a[middle])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



- **Test path** – a path (possibly with length 0) that starts at some initial node (N_0) and ends at some final node (N_f)
  - *path(t)* is the set of test paths corresponding to test case *t*
  - *path(T)* = {*path(t)* | *t* ∈ *T*}
  - Each test case gives at least one test path
    - If the program is deterministic, each test case gives exactly one path; i.e. the test case determines the test path
- Given a set of test requirements TR for a graph criterion C, a test set T satisfies C on graph G if & only if: for every test requirement tr in TR, at least one test path p in path(T) exists such that p satisfies tr

- Example:

```
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]

- **Nondeterminism**
  - Caused by dependence on inputs, the thread scheduler, or memory addresses (e.g. Java's hashCode())
  - More than one output might be a valid result of a single input

- **Statement coverage** – for each node n that can be reached by N_0, TR contains a requirement to visit n
  - i.e. T satisfies statement coverage if & only if for every syntactically reachable node n, there is some path in path(T) that visits n
  - Example:
    - We have TR = {n0, n1, n2, n3, n4, n5, n6} (these are nodes on a graph D)
    - path(t1) = p1 = [n0, n1, n3, n4, n6]
    - path(t2) = p2 = [n0, n2, n3, n5, n6]
    - Then T = {t1, t2} satisfies statement coverage on D
- **Branch coverage** – TR contains each reachable path of length ≤ 1 in G
  - These paths do not have to start in N_0
  - i.e. tests in TR must traverse every possible edge
- It's possible to get 100% coverage without testing anything – if test cases don't have asserts
- In real programs, 80% coverage is usually sufficient
- **Complete path coverage** – TR contains <u>all paths</u>
  - Impossible to achieve with cyclic graphs

Finite State Machines

- FSMs are higher-level than CFGs
  - They capture the design of the software
  - Generally there is no obvious mapping between FSM and code
  - Nodes = software states (e.g. sets of values for key variables)
  - Edges = transitions between software states (e.g. something changes in the environment; someone enters a command)
    - Often guarded by preconditions & postconditions
- Advantages of FSMs:
  - Enable creation of tests before implementation
  - Easier to analyze an FSM than the code
- Disadvantages of FSMs:
  - Abstract models are not necessarily exhaustive
  - Subjective (so they could be poorly done)
  - FSM may not match the implementation

- **Node/state coverage** – visit every FSM state
- **Edge/transition coverage** – traverse every FSM transition
- **Edge-pair/two-trip coverage** – extend edge coverage of paths with length ≤ 2

- **Round-trip path** – path of nonzero length with no internal cycles that starts and ends at the same node
- **Simple round-trip coverage (SRTC)** – TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path
  - i.e. if a node is part of a cycle, cover one of the cycles that the node is in
- **Complete round-trip coverage (CRTC)** - TR contains all round-trip paths for each reachable node in G
  - i.e. if a node is part of a cycle, cover all cycles that the node is in

- Deriving FSMs
  - Modelling state – choose relevant state variables and abstract them
  - Specifications – build FSM based on software structure; should resemble UML state charts

Syntax-Based Testing

- Applications of context-free grammars:
  - **Input-space grammars**: creating inputs (both valid and invalid)
  - **Program-based grammars**: modifying programs (mutation testing)
- Generating inputs: generate "valid" tests from regex and invalid tests by mutating the grammar/regex
  - But regex is limited in power (e.g. parentheses problem)
- Using input grammars:
  - **Recognizer** – can include them in a program to validate inputs
  - **Generator** – can create program inputs for testing
    - Begin with the start production and replace non-terminals with their right-hand sides to get (eventually) strings belonging to the input languages
- Example:

```
roll    = action*
action  = dep | deb
dep     = "deposit" account amount
deb     = "debit" account amount
account = digit { 3 }
amount  = "$" digit+ "." digit { 2 }
digit   = ["0" – "9"]
```

- Example mutation operators:
  - Non-terminal replacement

    dep = "deposit" account amount ⟹ dep = "deposit" amount amount
  - Terminal replacement

    amount = "$" digit+ "." digit { 2 } ⟹ amount = "$" digit+ "$" digit { 2 }
  - Terminal & non-terminal deletion

    dep = "deposit" account amount ⟹ dep = "deposit" amount
  - Terminal & non-terminal duplication

    dep = "deposit" account amount ⟹ dep = "deposit" account account amount
- **Fuzzing**
  - A useful technique for finding interesting test cases
  - Works best at interfaces between components
  - But without significant work, won't find sophisticated domain-specific issues
  - **Generation-based fuzzing** – starts with a grammar and generates inputs that match the grammar
    - Generate random strings from the grammar, then feed as input and look for assertion failures

- E.g. generating C programs:
    1. sequence of ASCII characters;
    2. sequence of words, separators, and white space (gets past the lexer);
    3. syntactically correct C program (gets past the parser);
    4. type-correct C program (gets past the type checker);
    5. statically conforming C program (starts to exercise optimizations);
    6. dynamically conforming C program;
    7. model conforming C program.
  - Each of these levels contains a subset of the inputs from previous levels. However, as the level increases, we are more likely to find interesting bugs that reveal functionality specific to the system (rather than simply input validation issues).
- **Mutation-based fuzzing** – starts with existing test cases and randomly modifies them to explore new behaviours
  - E.g. randomly flip bytes, or parse input and change non-terminals
  - A finite given input set can only be fuzzed into a limited set of possibilities & code paths; will need to provide new input sets at that point

Mutation Testing

- **Ground string** – a (valid) string belonging to the language of the grammar
- **Mutation operator** – a rule that specifies syntactic variations of strings generated from a grammar
- **Mutant** – the result of one application of a mutation operator to a ground string
  - Only apply one operator
  - Are valid programs that ought to behave differently than the ground string
- Use mutated grammar to generate invalid test strings
  - Can generate strings still in the grammar (valid) even after mutation
  - Some programs accept only a subset of a specified larger language, e.g. Blogger HTML comments; then testing intersection is useful
- Given a mutant m generated from ground string m_0
  - Test case t (strongly) kills m if running t on m gives different output than running t on m_0
  - Once the mutant is found, we keep the test case
  - Mutation score = percentage of mutants killed
- Example:

```
int foo(int x, int y) { // original
  if (x > 5) return x + y;
  else return x;
}
```

```
int foo(int x, int y) { // mutant
  if (x > 5) return x - y;
  else return x;
}
```

```
// original
int min(int a, int b) {
  int minVal;
  minVal = a;

  if (b < a) {


    minVal = b;




  }
  return minVal;
}
```

```
// with mutants
int min(int a, int b) {
  int minVal;
  minVal = a;
  minVal = b;                // Δ 1
  if (b < a) {
  if (b > a) {               // Δ 2
  if (b < minVal) {          // Δ 3
    minVal = b;
    BOMB();                  // Δ 4
    minVal = a;              // Δ 5
    minVal = failOnZero(b); // Δ 6
  }
  return minVal;
}
```

- Uninteresting mutants:
  - **Stillborn** – cannot compile/immediately crash
  - **Trivial** – killed by almost any test case
  - **Equivalent** – indistinguishable from original program
- **Strong mutation** – fault must be reachable, infect state, and propagate to output (i.e. a failure)
  - **Strong mutation coverage** – for each mutant m, TR contains a test which strongly kills m
- **Weak mutation** – fault which kills a mutant need only be reachable and infect state (i.e. an error)
  - **Weak mutation coverage** – for each mutant m, TR contains a test which weakly kills m

- For the above example (Δ1),
  - reachability: unavoidable;
  - infection: need $b \neq a$;
  - propagation: wrong `minVal` needs to return to the caller; that is, we can't execute the body of the `if` statement, so we need $b > a$.

  - Test case for strong mutation: a = 5, b = 7
  - Test case for weak mutation: a = 7, b = 5
- Δ3 in the above example is an equivalent mutant

- **Integration mutation** – mutate interfaces between methods
  - Change calling method by changing actual parameter values
  - Change calling method by changing callee
  - Change callee by changing inputs and outputs
- **OO program mutation** – modify object fields & methods

	Program-based	Input Space/Fuzzing
Grammar	Programming language	Input languages / XML
Summary	Mutates programs / tests integration	Input space testing
Use Ground String?	Yes (compare outputs)	Sometimes
Use Valid Strings Only?	Yes (mutants must compile)	No
Tests	Mutants are not tests	Mutants are tests
Killing	Generate tests by killing	Not applicable

- Is mutation testing any good?
  - Yes: test suites that kill more mutants are also better at finding real bugs
- Is graph coverage any good?
  - Coverage does not correlate with high quality when it comes to test suites
  - Specifically: test suites that are larger are better because they are larger, not because they have higher coverage
  - Furthermore, stronger coverage (e.g. branch vs statement, logic vs branch) doesn't result in better test suites

Test Design Principles

- Many small tests, not one big test
- Make it easy to add new tests
- Unit vs. integration tests
- Write some code, write some tests, repeat
- Flaky tests are terrible
- Look inside the system under test