INFR10057 - Software Testing

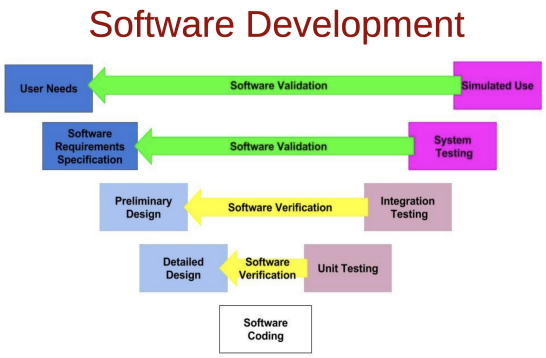
Revision Notes

Table of Contents

Lecture 1 - Testing Overview

Why? Software is growing in size and complexity

V-model of Software Development



Software Requirements and Specification

Example:

Requirement: Valid students can login to Euclid

Specifications

- Student is valid if he/she is associated with the valid login and password
  - Valid login: s + 6 digits not more than 4 years old
  - Login and password should match in the database

Validation and Verification

Validation: are we building the right product? Does the software implement customer requirements?

Verification: are we building the product right? Does the implementation implement the specification?

Software Bugs

Example IT hiccups:

- Airlines - delays, air traffic computer problems, reservation systems, tracking systems, crew scheduling, network failure
- Automarkers - steering problem, cruise control, stalling-related issues, coding error in spot-welding robots
- Communications - software problems in 3G and 4G network, expired software certificate
- Etc.

Sources of Problems

Requirements definitions: erroneous, incomplete, inconsistent requirements

Design: fundamental design flaws in the software

Implementation: mistakes in chip fabrication, wiring, programming faults, malicious code

Support systems: poor programming languages, faulty compilers and debuggers, misleading development tools

Specification - “If you can’t say it, you can’t do it”

- You have to know what your product is before you can say if it has a bug
- A *specification* defines the product being created and includes:
    - Functional requirements thad describes the features the product will support
        - E.g. on a word processor: save, print, check spelling, change font
    - Non-functional requirements are constraints on the product
        - E.g. security, reliability, user friendliness, platform

Example: cruise control system

Inputs: Brake Pressed, Desired Road Inclination, Wind resistance, Throttle position

Output: Throttle

Sample input range:

Desired speed = 0 to 200 km/h

Road inclination = -30% to +30%

Wind resistance = 0 to 50 km/h

Throttle position = 0.0 to 5.0

Sample non-functional requirements: 99% reliability

A software bug occurs when at least one of these rules are true

- The software does not do something that the specification says it should do
- The software does something that the specification says it should not do
- The software does something that the specification does not mention
- The software does not do something that the product specification does not mention but should
- The software is difficult to understand, hard to use, slow, etc.

Most bugs are not because of mistakes in code…

- Specification (~=55%)
- Design (~=25%)
- Code (~=15%)
- Other (~=5%)

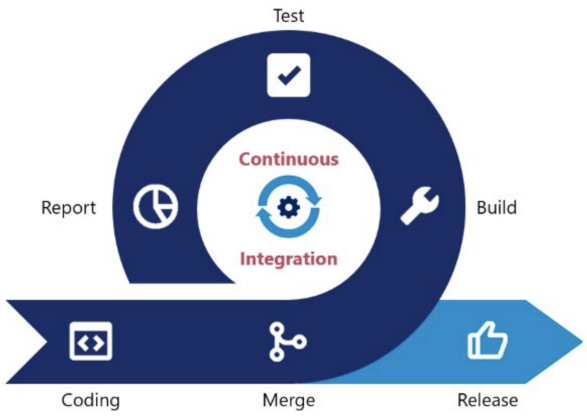
Relative cost of bugs - “bugs found later cost more to fix”

- Cost to fix a bug increases exponentially (10x)
    - I.e. it increases tenfold as time increases
- E.g. a bug found during specification costs $1 to fix
    - If it was found in design, cost = $10
    - If it was found in code, cost = $100
    - If it was found in released software, cost = $1000

Goal of a software tester

- To find bugs as early in the software development processes as possible and make sure they get fixed
- Advice: be careful not to get caught in the dangerous spiral of unattainable perfection

Developing large software



Testing at scale

“In an average day, TAP integrates and tests at enormous compute cost - more than 13L code projects, requiring 800K builds and 150 million test runs.”

Tasks involved in Testing

Automation is where the future of testing is.

Test generation

- You have a large input space, with a different range for each input
- Exhaustive search space is huge
- How do you sample this input space without testing every possible value? Which inputs are important?

Automated Test Generation

1. Covers input/output space
    a. Property-based testing
    b. Combinatorial testing
    c. Fuzz testing
2. Covers structure of program or model - make sure every true/false branch in my if statement is covered
    a. Model-based testing
    b. Coverage-based testing
    c. Mutation testing - introduce artificial false and check if you have tests that will catch these artificial falses
    d. Symbolic and concolic testing - state space is too large, can I abstract it to a smaller state space?

Test Adequacy

Testing is an incomplete technique; you cannot provide guarantees with testing.

Test adequacy criteria

- Criteria that identify inadequacies in test suites - based on specification, code, model, simulated bugs
- Code coverage metrics are commonly used to assess thoroughness of test suites

Test Execution

Automated execution with build and test frameworks - Jenkins, GoogleTest, Junit, Maven

Test Oracle

A way of checking if your tests pass or fail.

Test Oracle Problem

- Challenge of automatically determining the correctness of test executions

Types of test oracles

- Specified oracle - judges behavioural aspects with respect to formal specifications
- Implicit oracle - help reveal particular anomalies in executions like buffer overflows, segmentations faults, etc.
- Derived oracle - use documentations or system specification to judge a system's behaviour, e.g. metamorphic relations and inferring invariants (e.g. autocars that detect pedestrians regardless of weather/visibility due to weather)

Important Terminology and Concepts

- Software fault: a static defect in software (cause)
- Software error: an incorrect internal state that is the manifestation of some fault (symptoms)
- Software failure: external, incorrect behaviour with respect to the requirements or other description of the expected behaviour

Understanding the difference will help you fix faults. Write your code so that it is testable.

Fault/Error/Failure Example

```
int count.spaces(char* str) {
    int length, i, count;
    count = 0;
    length = strlen(str);
    for(i = 1; i < length; i++) {
        if (str[i] === ' ') { count++; }
    }
    return(count);
}
```

Software fault: `i = 1` should be `i = 0`

Software error: some point in the program where you incorrectly count the number of spaces

Failure inputs and outputs that can make the fault happen

- `count.spaces("H H H")` would not cause the failure while `count.spaces(" H")` does

Unit Testing

- xUnit testing is a framework where individual functions and methods are tested
- It is not particularly well-suited to integration testing or regression testing

- The best way to write testable code is to write the tests as you develop the code
- Writing the test cases after the code takes more time and effort than writing the test during the code - it is like good documentation; you'll always find something else to do if you leave until after you've written the code

Lecture 2 - Tools for Unit Test: JUnit

Continuous Integration

- Small bits of code
- Merge into version control
- Build the code
- Test the code you just merged (existing tests or adding new tests)
    - Unit testing: testing the smallest component of your code (e.g. functions/classes)

JUnit

JUnit is a framework for writing tests.

- Written by Erich Gamma (Design Patterns) and Kent Beck (eXtreme Programming)
- JUnit uses *Java's reflection capabilities* (Java programs can examine their own code) and (as of version 4) *annotations*
- JUnit allows us to
    - Define and execute tests and test suites
    - Use test as an effective means of specification
    - Write code and use the tests to support refactoring
    - Integrate revised code into a build
- JUnit is available on several IDEs, e.g. BlueJ, JBuilder, and Eclipse have JUnit integration to some extent

JUnit's Terminology

- A **test runner** is software that runs tests and reports results
    - Many implementations: standalone GUI, command line, integrated into IDE
- A **test suite** is a collection of test cases
- A **test case** tests the response of a single method to a particular set of inputs
- A **unit test** is a test of the smallest element of code you can sensibly test, usually a single class
- A **test fixture** is the environment in which a test is run
    - A new fixture is set up before each test case is executed, and torn down afterwards
    - Example: if you are testing a database client, the fixture might place the database server in a standard initial state, ready for the client to connect
- An **integration test** is a test of how well classes work together
    - JUnit provides some limited support for integration tests
- *Proper* unit testing would involve **mock objects** - fake versions of the other classes with which the class under test interacts
    - JUnit does not help with this - it is worth knowing about, but not always necessary

Structure of a JUnit (4) test class

We want to test a class named `Triangle`.

- This is the unit test for the `Triangle` class; it defines objects used by one or more tests.

```
public class TriangleTest { }
```

- This is the default constructor

```
public TriangleTest()
```

- This is how to create a test fixture by creating and initialising objects and values (set-up)

```
@Before public void init()
```

- This is how to release any system resources used by the test fixture. Java usually does this for free, but files, network connections, etc. might not get tidied up automatically. (tear-down)

```
@After public void cleanUp()
```

- These methods contain tests for the `Triangle` constructor and its `isScalene()` method

```
@Test public void noBadTriangles()
@Test public void scaleneOk()
```

Making Tests: Assert

- Within a test
  - Call the method being tested and get the actual result
  - *Assert* a property that should hold of the test result
  - Each *assert* is a challenge on the test result
- If the property fails to hold then `assert` fails, and throws an `AssertionFailedError`:
  - JUnit catches these errors, records the results of the test and displays them
- The following throws an `AssertionFailedError` if the test fails. The optional message is included in the error.

```
static void assertTrue(boolean test)
static void assertTrue(String message, boolean test)
```

- The following throws an `AssertionFailedError` if the test succeeds. The optional message is included in the error.

```
static void assertFalse(boolean test)
static void assertFalse(String message, boolean test)
```

- This method is heavily overloaded: *expected* and *actual* must be both objects or both of the same primitive type. For objects, use your equals method, if you have defined it properly, as `public boolean equals(Object o)` - otherwise it uses `==`.

```
assertEquals(expected, actual)
assertEquals(String message, expected, actual)
```

- This method asserts that two objects refer to the same object (using `==`).

```
assertSame(Object expected, Object actual)
assertSame(String message, Object expected, Object actual)
```

- This method asserts that two objects do not refer to the same object (using `==`).

```
assertNotSame(Object expected, Object actual)
assertNotSame(String message, Object expected, Object actual)
```

- This method asserts that the object is null

```
assertNull(Object object)
assertNull(String message, Object object)
```

- This method asserts that the object is not null.

```
assertNotNull(Object object)
assertNotNull(String message, Object object)
```

- This method causes the test to fail and throw an `AssertionFailedError` - useful as a result of a complex test, when the other assert methods are not quite what you want.

```
fail()
fail(String message)
```

The `assert` statement in Java

When to use the `assert` statement:

- Use it to document a condition that you 'know' to be true
- Use `assertFalse` in code you 'know' cannot be reached (such as a default case in a switch statement)
- Do not use `assert` to check whether parameters have legal values, or other places where throwing an Exception is more appropriate
- **Can be dangerous:** customers are not impressed by a library bombing out with an assertion failure.

Lecture 3 - Functional Testing

Functional Testing

- Deriving test cases from program specifications
  - Functional refers to the source of information used in test case design, not to what is tested
- Also known as
  - Specification-based testing (from specifications)
  - Black-box testing (no view of the code)
- Functional specification = description of intended program behaviour
  - Either formal or informal
- Example of functional specification
  - Specification 1: execute bank transfer from account A to account B for amount *tr_amt* if account A is verified, bank *transfer* function is selected, account B is verified, and *tr_amt* < balance in account A.

Systematic vs Random Testing

- Random (uniform)
  - Pick possible inputs uniformly
  - Avoids designer bias
    - A real problem: the test designer can make the same logical mistakes and bad assumptions as the program designer (especially if they are the same person)
  - But treats all inputs are equally valuable
- Systematic (non-uniform)
  - Try to select inputs that are especially valuable
  - Usually by choosing representatives of classes that are apt to fail often or not at all
  - Functional testing is systematic

Why not Random Testing?

- The distribution of bugs is not uniform
- Pareto principle: it is typical that 80% of the bugs in your programs is usually in 20% of the modules
- Example: Java class 'roots' applies quadratic equation

```
x = [(-b + Math.sqrt(b^2 - 4ac) / 2*a), (-b - Math.sqrt(b^2 - 4ac) / 2*a)]
```

  - Incomplete implementation logic: program does not properly handle the case in which `(b^2 - 4ac) = 0` and `a = 0`
  - Failing values are sparse in the input space - needles in a very big haystack
    - Random sampling is unlikely to choose `a = 0.0` and `b = 0.0`
- Random testing treats all values as the same
- Example of random testing: Randoop

Consider the purpose of testing

- To estimate the proportion of needles to hay, sample randomly

    - Reliability estimation requires unbiased samples for valid statistics (not our goal!)
    - What is the proportion of failures with respect to all runs?
- To find needles and remove them from hay, look systematically (non-uniformly) for needles
    - Unless there are a lot of needles in the haystack, a random sample will not be effective at finding them
    - We need to use everything we need to know about the needles (are they heavier than hay? Do they sift to the bottom?

Systematic Partition Testing

- Take the space of the input (space of behaviours) and try to partition them
- Failures are sparse in the space of possible inputs but dense in some parts of the space
- If we systematically test some cases from each part, we will include the dense parts
- Functional testing is one way of drawing pink lines to isolate regions with likely failures

The Partition Principle

- Exploit some knowledge to choose samples that are more likely to include 'special' or trouble-prone regions of the input space
    - Failures are sparse in the whole input space
    - But we may find regions in which they are dense
- (Quasi-)Partition testing: separates the input space into classes whose union is the entire space
    - Quasi because the classes may overlap
- Desirable case: each fault leads to failures that are dense (easy to find) in some class or inputs
    - Sampling each class in the quasi-partition selects at least one input that leads to a failure, revealing the fault
    - Seldom guaranteed; we need to depend one experience-based heuristics

Example: password field

Valid passwords

- 6-10 characters long
- With at least 1 uppercase character, 1 lowercase character and 1 number

Partitions

- Lengths: <6, =6, 6-10, =10, >10
- Content: no uppercase, 1 uppercase, no lowercase, 1 lowercase, no number, 1 number, etc. (different combinations of uppercase, lowercase and number)

Functional testing: exploiting the specification

- Functional testing uses the specification (formal or informal) to partition the input space
    - E.g. specification of 'roots' program suggests division between cases with zero, one, and two real roots
- Test each category, and boundaries between categories
    - No guarantees, but experience suggests failures often lie at boundaries

Why functional testing?

- The baseline technique for designing test cases
    - Timely
        - Often useful in refining specifications and assessing testability before code is written
    - Effective
        - Finds some classes of fault (e.g. missing logic) that can elude other approaches
    - Widely applicable
        - To any description of program behaviour serving as specification
        - At any level of granularity from module to system testing
    - Economical
        - Typically less expensive to design and execute than structural (code-based) test cases
        - Find bugs earlier in the development stage

Early functional test design

- Program code is not necessary
    - Only a description of intended behaviour is needed
    - Even incomplete and informal specifications can be used
        - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
    - Often reveals ambiguities and inconsistency in specification
    - Useful for assessing testability
        - And improving test schedule and budget by improving specification
    - Useful explanation of specification
        - Or in extreme case (as in extreme programming), test cases are the specifications

Functional versus Structural: Classes of faults

- Different testing strategies (functional, structural, fault-based, model-based) are most effective for different classes of faults
- Functional testing is best for missing logic faults
    - A common problem: some program logic was simply forgotten
    - Structural (code-based) testing will never focus on code that isn't there

Functionality vs Structural test: Granularity levels

- Functional test applies at all granularity levels
    - Unit (from module interface spec, test the modules)
    - Integration (from API or subsystem spec, test interactions between modules)
    - System (from system requirements spec, test the entire system with all the modules)
    - Regression (from system requirements + bug history, for changes within a program)
- Structural (code-based) test design applies to relatively small parts of a system
    - Unit
    - integration

Steps: From specification to test cases

1. Decompose the specification
    a. If the specification is large, break it down into **independently testable features** to be considered in testing
        i. Simple behaviours which you can test
2. Select representatives
    a. Representative values of each input, or
    b. Representative values of a model
        i. Often simple input/output transformations don't describe a system
        ii. We use models in program specification, in program design, and in test design
3. Form test specifications
    a. Typically: combinations of input values, or model behaviours
4. Produce and execute actual tests

Simple example: postal code lookup

Input: ZIP code (5-digit US postal code)

Output: List of cities

Representative values

- Correct zip code: with 0, 1, or many cities
- Malformed zip code: empty (1-4 characters, 6 characters, very long), non-digit characters, non-character data

Lecture 4 - Combinatorial Testing

Basic Idea

- Identify distinct attributes that can be varied
    - In the data, environment, or configuration
    - Example: browser could be “IE” or “Firefox”, operating system could be “Vista”, “XP” or “OSX”
- Systematically generate combinations to be tested
    - Example: IE on Vista, IE on XP, Firefox on Vista, Firefox on OSX
- Rationale: Test cases should be varied and include possible “corner cases”
- Go through parameters, see what the values are, enumerate every possible combinations and create your test specifications

Category partition (manual steps)

1. Decompose the specification into independently testable features
    a. For each feature identify parameters and environment elements
    b. For each parameter and environment element identify elementary characteristics (categories)
2. Identify relevant values
    a. For each characteristic (category) identity (classes of) values
        i. Normal values
        ii. Boundary values
        iii. Special values
        iv. Error values
3. Introduce constraints (automatic only in catalog-based testing)
    a. A combination of values for each category corresponds to a test case specification
    b. Introduce constraints to
        i. Rule out impossible combinations
        ii. Reduce the size of the test suite if too large
    c. Error constraints
        i. [Error] indicates a value class that
            - Corresponds to erroneous values
            - Need to be tried only once
        ii. Error value classes
            - No need to test all possible combinations of errors
            - One test is enough (we assume that handling an error case bypasses other program logic)
    d. Single constraints
        i. Indicates a value class that test designers choose to test only once to reduce the number of test cases
        ii. Note: single and error have the same effect but differ in rationale; keeping them distinct is important for documentation and regression testing

Pairwise combinatorial testing

- Category partition works well when intuitive constraints reduce the number of combinations to a small amount of test cases
    - Without many constraints, the number of combinations may be unmanageable
- Pairwise combination (instead of exhaustive)
    - Generate combinations that efficiently cover all pairs (triples, …) of classes
    - Rationale: most failures are triggered by single values or combinations of a few values. Covering pairs (triples, …) reduces the number of test cases, but reveals most faults
- Make sure are two-way combinations are tested

Catalog-based testing

- Deriving value classes requires human judgement
- Gathering experience in a systematic collection can:
    - Speed up the test design process
    - Routinise many decisions, better focusing on human effort
    - Accelerate training and reduce human error
- Catalogs capture the experience of test designers by listing important cases for each possible type of variable
    - Example: if the computation uses an integer variable a catalog might indicate the following relevant cases
        - The element immediately preceding the lower bound
        - The lower bound of the interval
        - A non-boundary element within the interval
        - The upper bound of the interval

Catalog-based testing process

1. Analyse the initial specification to identify simple elements:
    a. Pre-conditions
    b. Post-conditions
    c. Definitions
    d. Variables
    e. Operations
2. Derive a first set of test case specifications from pre-conditions, post-conditions and definitions
3. Complete the set of test case specifications using test catalogs (take existing specs and enumerate each of these values for the different variables and increase the number of specs you have for testing)

What we have so far

- From category partition testing
    - Division into a (manual) step of identifying categories and values, with constraints, and an (automated) step of generating combinations
- From catalog based testing

    - Improving the manual step by recording and using standard patterns for identifying significant values
- From pairwise testing
    - Systematic generation of smaller test suites
- These ideas can be combined

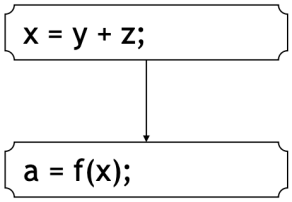
Lecture 5 - Finite Models

Properties of Models

- **Compact:** representable and manipulable in a reasonably compact form
  - What is *reasonably compact* depends largely on how the model will be used
- **Predictive:** must represent some salient characteristics of the modeled artifact well enough to distinguish between *good* and *bad* outcomes of analysis
  - No single model represents all characteristics well enough to be useful for all kinds of analysis
- **Semantically meaningful:** it is usually necessary to interpret analysis results in a way that permits diagnosis of the causes of failure
- **Sufficiently general:** models intended for analysis of some important characteristic must be general enough for practical use in the intended domain of application

Graph Representations

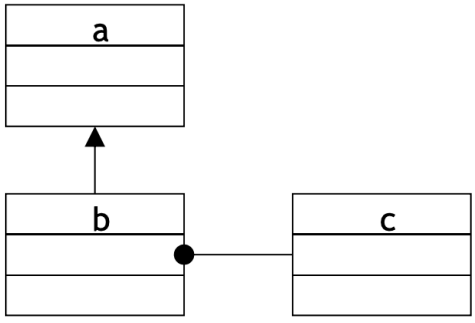
- Directed graph
  - N (set of nodes)
  - E (relation on the set of nodes) edges
- Labels and code
  - We can label nodes with the names or descriptions of the entities they represent
    - If node a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a, b) connecting them in this way:



- Multidimensional Graph Representations
  - Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once
    - Class B extends (is a subclass of) class A
    - Class B has a field that is an object of type C

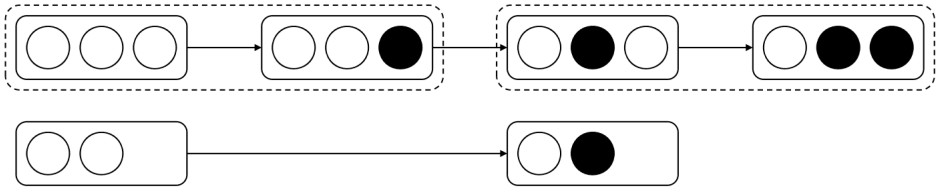
extends relation
NODES = {A, B, C}
EDGES = {(A,B)}

includes relation
NODES = {A, B, C}
EDGES = {(B,C)}

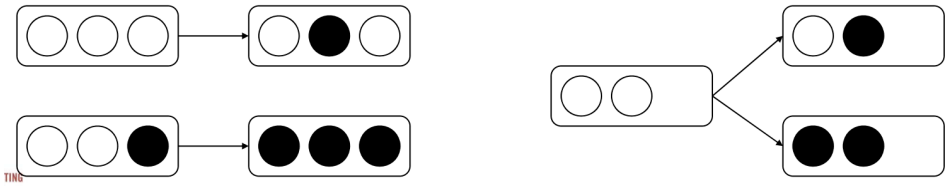


Finite Abstraction of Behavior

An abstraction function suppresses some details of program execution...



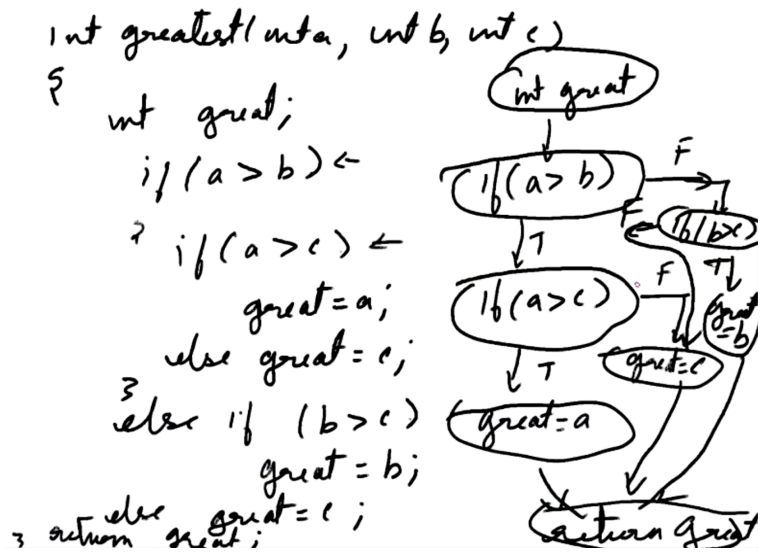
...it lumps together execution states that differ with respect to the suppressed details but are otherwise identical



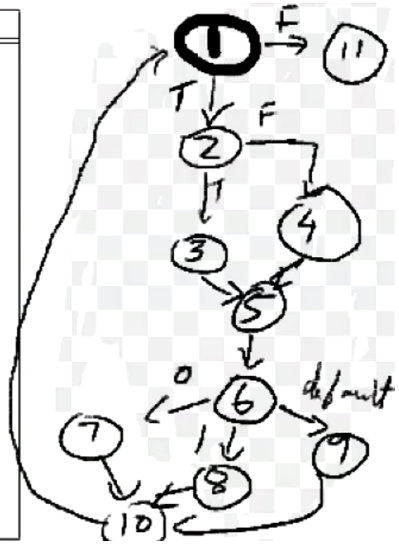
(Intraprocedural) Control Flow Graph

- Nodes = regions of source code (basic blocks)
  - Basic block = maximal program region with a single entry and single exit point
  - Often statements are grouped in single regions to get a compact model
  - Sometimes single statements are broken into more than one node to model control flow within the statement
- Directed edges = possibility that program execution proceeds from the end of one region directly to the beginning of another
- Most of the time used only within a function

CFG Example from lecture

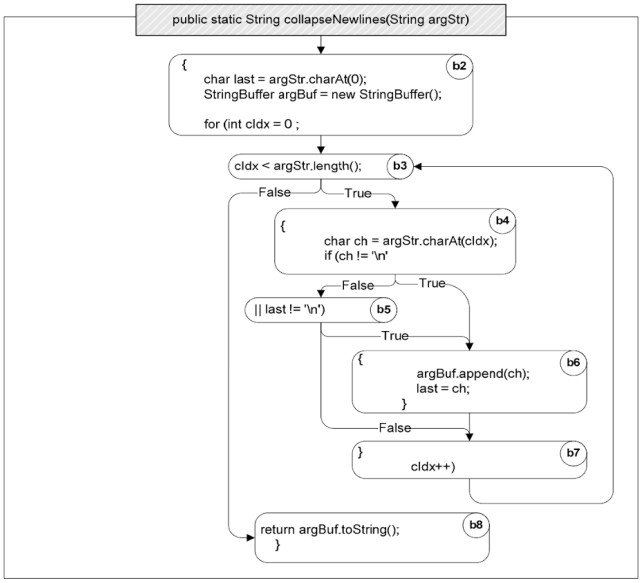


Node	Statement
(1)	while(x<100){
(2)	if (a[x] % 2 == 0) {
(3)	parity = 0;
	}
	else {
(4)	parity = 1;
(5)	}
(6)	switch(parity){
	case 0:
(7)	println("a[" + i + "] is even");
	case 1:
(8)	println("a[" + i + "] is odd");
	default:
(9)	println("Unexpected error");
	}
(10)	x++;
	}
(11)	p = true;



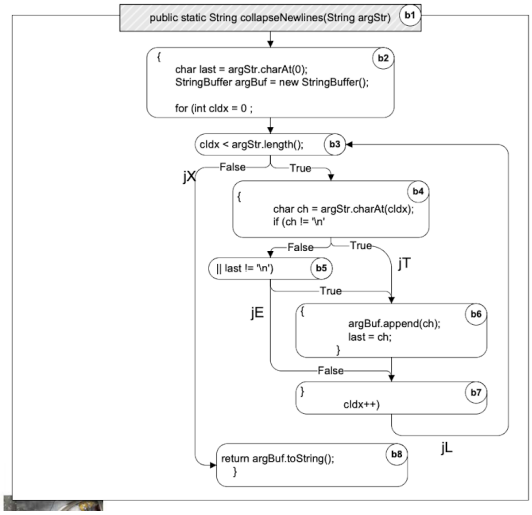
CFG Example from slides

```
public static String collapseNewlines(String argStr)
  {
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();

    for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
    {
      char ch = argStr.charAt(cIdx);
      if (ch != '\n' || last != '\n')
      {
        argBuf.append(ch);
        last = ch;
      }
    }

    return argBuf.toString();
  }
```



Linear Code Sequence and Jump (LCSJ)

Essentially subpaths of the control flow graph from one branch to another



From	Sequence of basic blocs	To
Entry	b1 b2 b3	jX
Entry	b1 b2 b3 b4	jT
Entry	b1 b2 b3 b4 b5	jE
Entry	b1 b2 b3 b4 b5 b6 b7	jL
jX	b8	ret
jL	b3 b4	jT
jL	b3 b4 b5	jE
jL	b3 b4 b5 b6 b7	jL

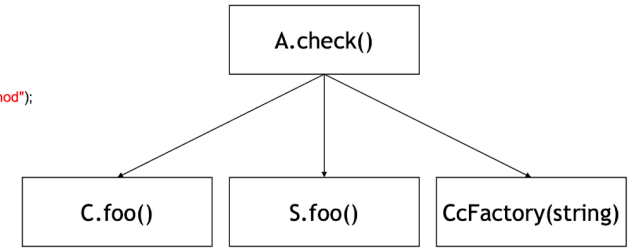
Interprocedural control flow graph

- Call graphs
  - Nodes represent procedures
    - Methods
    - C functions
  - Edges represent calls relation

Overestimating the *calls* relation

The static call graph increases calls through dynamic bindings that never occur in execution.

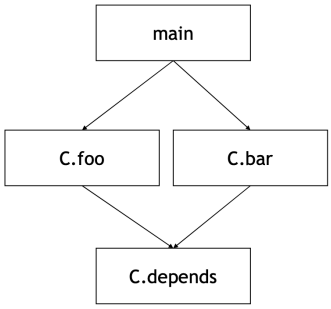
```
public class C {
    public static C cFactory(String kind) {
        if (kind == "C") return new C();
        if (kind == "S") return new S();
        return null;
    }
    void foo() {
        System.out.println("You called the parent's method");
    }
    public static void main(String args[]) {
        (new A()).check();
    }
}
class S extends C {
    void foo() {
        System.out.println("You called the child's method");
    }
}
class A {
    void check() {
        C myC = C.cFactory("S");
        myC.foo();
    }
}
```



Context Insensitive Call graphs

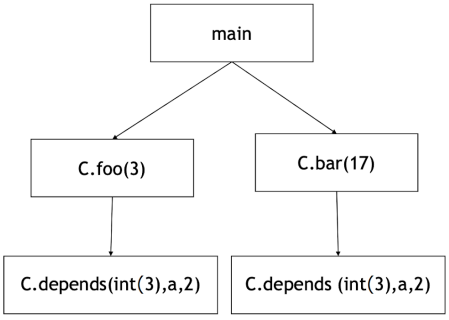
You are agnostic to the arguments passed through your method calls and who called you.

```
public class Context {
    public static void main(String args[]) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }

    void foo(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 2) ;
    }

    void bar(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 16) ;
    }

    void depends( int[] a, int n ) {
        a[n] = 42;
    }
}
```

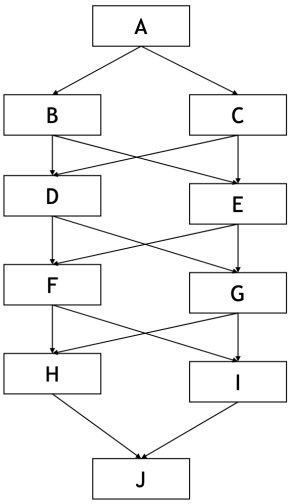


Context Sensitive Call graphs

```
public class Context {
    public static void main(String args[]) {
        Context c = new Context();
        c.foo(3);
        c.bar(17);
    }

    void foo(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 2) ;
    }

    void bar(int n) {
        int[]  myArray = new int[ n ];
        depends( myArray, 16) ;
    }

    void depends( int[] a, int n ) {
        a[n] = 42;
    }
}
```



Context Sensitive CFG Exponential Growth



1 context A

2 contexts AB AC

4 contexts ABD ABE ACD ACE

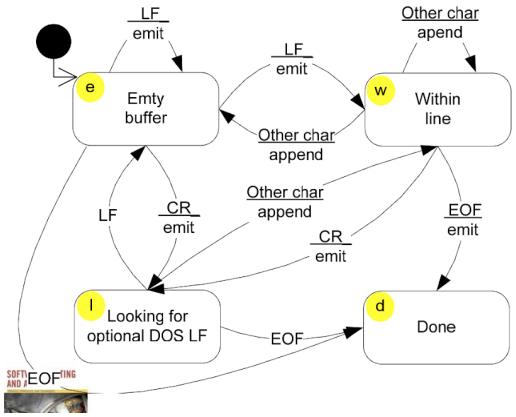
8 contexts …

16 calling contexts …

Finite State Machines

- Finite set of states (nodes)
- Set of transitions among states (edges)

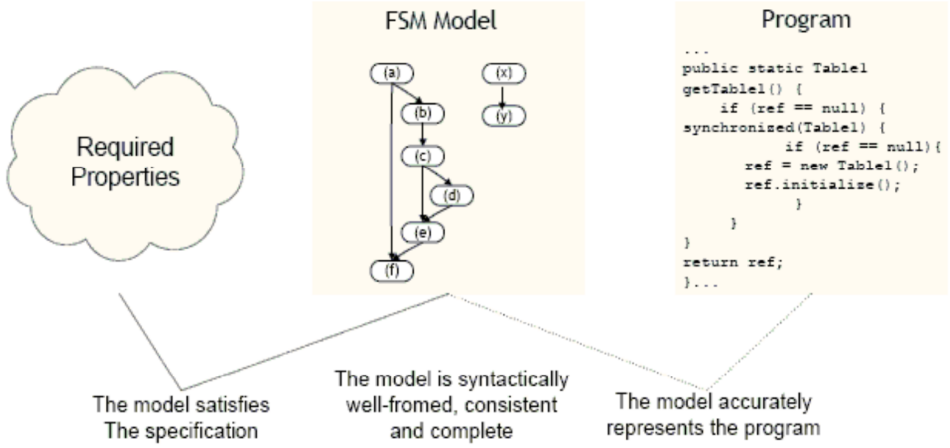
Graph representation (Mealy machine)



Tabular representation

	LF	CR	EOF	other
e	e/emit	e/emit	d/-	w/append
w	e/emit	e/emit	d/emit	w/append
l	e/-		d/-	w/append

Using Models to Reason about System Properties



Summary

- Models must be much simpler than the artifact they describe to be understandable and analysable
- Must also be sufficiently detailed to be useful
- CFG are built from software
- FSM can be built before software to document intended behavior

Lecture 6 - Structural Testing: Part I

"Structural" Testing

- Judging test suite thoroughness based on the structure of the program itself
  - Also known as "white-box", "glass-box" or "code-based" testing
  - To distinguish from functional (requirements-based, "black-box" testing)
    - "Structural" testing is still testing product functionality against its specification
    - Only the measure of thoroughness has changed

Why Structural (code-based) testing?

- One way of answering the question "What is missing in our test suite?"
  - This part of your code won't be tested with functional tests
  - If part of a program is not executed by any test case in the suite, faults in that part cannot be exposed
  - But what is a 'part'?
    - Typically, a control flow element or combination
      - Statements (of CFG nodes), branches (or CFG edges)
      - Fragments and combinations: conditions, paths
- Complements functional testing: another way to recognise cases that are treated differently
  - Recall functional fundamental rationale: prefer test cases that are treated differently over cases treated the same

No guarantees

- Executing all control flow elements does not guarantee finding all faults
  - Execution of a faulty statement may not always result in a failure
    - The state may not be corrupted when the statement is executed with some data values
    - Corrupt state may not propagate through execution to eventually lead to failure
- What is the value of structural coverage?
  - Increases confidence in thoroughness of testing
    - Removes some obvious inadequacies

Structural testing *complements* functional testing

- There might be cases where your specification is incomplete, or implementation where there are several cases for one specification, and you need to test that with structural testing
- Control flow testing includes cases that may not be identified from specifications alone
  - Typical case: implementation of a single item of the specification by multiple parts of the program
  - Example: hash table collision (invisible in interface spec)
- Test suites that satisfy control flow adequacy criteria could fail in revealing faults that can be caught with functional criteria
  - Typical case: missing path faults

Structural testing in practice

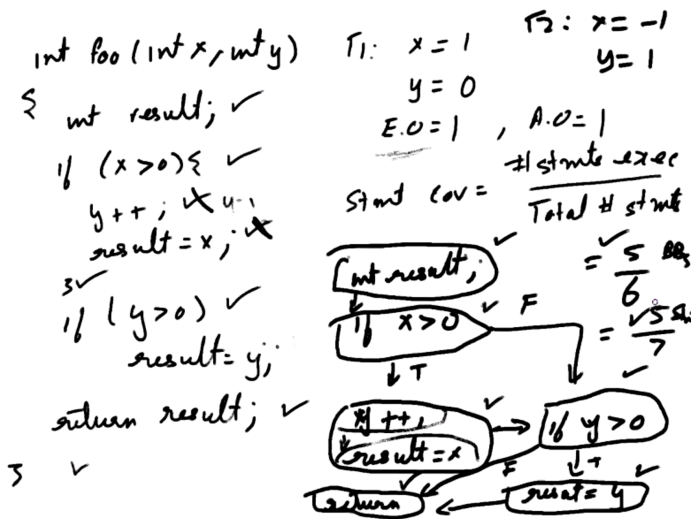
- Create a functional test suite first, then measure structural coverage to identify what is missing
- Interpret unexecuted elements
  - May be due to natural differences between specification and implementation
  - Or may reveal flaws of the software or its development process

    - Inadequacy of specifications that do not include cases present in the implementation
    - Coding practice that radically diverges from the specification
    - Inadequate functional test suites
- Attractive because automated
    - Coverage measurements are convenient progress indicators
    - Sometimes used as a criterion of completion
        - Use with caution: does not ensure effective test suites

Statement testing

- Adequacy criterion: each statement (or node in the cFG) must be executed at least once
- Coverage = no. of executed statements / total no. of statements
- Rationale: a fault in a statement can only be revealed by executing the faulty statement

Example from lecture



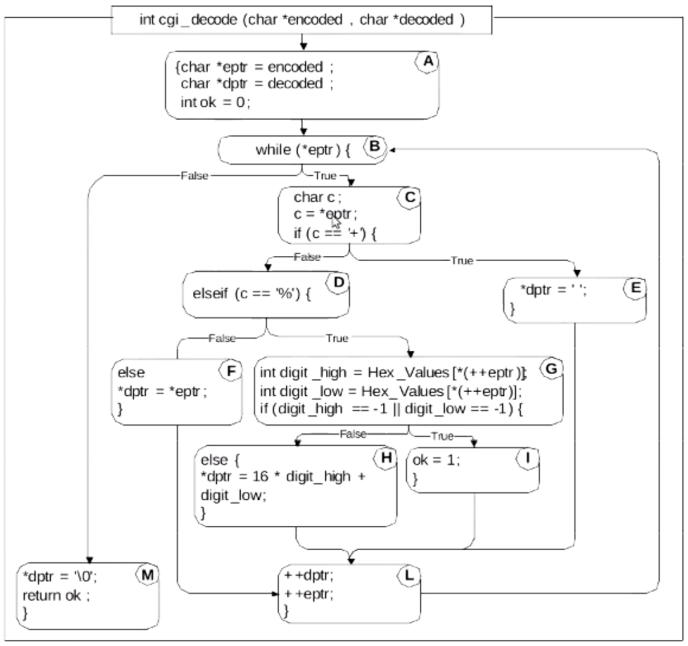
- T1 achieves 100% statement coverage
- Statement coverage is a weak criterion that is not effective in catching bugs
- If x < 0 and y < 0, the function will return an uninitialised result
    - Statement coverage will not catch this!
    - Data flow testing will catch this bug
- If the developer mistakenly returned x instead of result and x =1, y = 0, and the expected output = actual output = 1, therefore this bug will not be caught by statement coverage
- Conclusion: coverage criterion might not be rigorous

Example from slides

T_0 =
{"", "test",
"test+case%1Dadequacy"}
17/18 = 94% Stmt Cov.

T_1 =
{"adequate+test%0Dexecuti
on%7U"}
18/18 = 100% Stmt Cov.

T_2 =
{"%3D", "%A", "a+b",
"test"}
18/18 = 100% Stmt Cov.



Statements or blocks?

- Nodes in a control flow graph often represent basic blocks of multiple statements
  - Some standards refer to basic block coverage or node coverage
  - Difference in granularity, not in concept
- No essential difference
  - 100% node coverage <-> 100% statement coverage
    - But levels will differ below 100%
  - A test case that improves one will improve the other
    - Though not by the same amount, in general

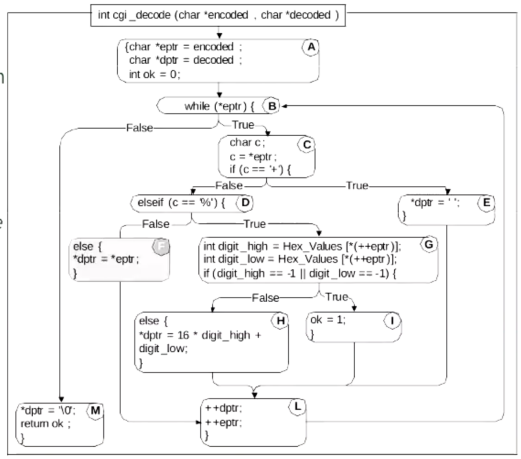
Coverage is not size

- More tests != more coverage!
- Coverage does not depend on the number of test cases
- Minimising test suite size is seldom the goal
  - Small test cases make failure diagnosis easier
  - In the example from the slides, a failing test case in T2 gives more information for fault localisation than a failing test case in T1

“All statements” can miss some cases

- Complete statement coverage may not imply executing all branches in a program
- Example:
  - Suppose block F were missing
  - Statement adequacy would not require *false* branch from D to L

T_3 =
{“”, “+%0D+%4J ”}
100% Stmt Cov.
No *false* branch from D



Branch testing

- Adequacy criterion: each branch (edge in the CFG) must be executed at least once
- Coverage: no. of executed branches / total no. of branches

Statements vs branches

- Traversing all edges of a graph causes all nodes to be visited
  - So test suites that satisfy the branch adequacy criterion for a program P also satisfy the statement adequacy criterion for the same program
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

“All branches” can still miss conditions

- Sample fault: missing operator (negation)

```
digit_high == 1 || digit_low == -1
```

- Branch adequacy criterion can be satisfied by varying `digit_low`
  - The faulty sub-expression might never determine the result

- We might never really test the faulty condition, even though we tested both outcomes of the branch

Condition Testing

- Branch coverage exposes faults in how a computation has been decomposed into cases
  - Intuitively attractive: check the programmer's case analysis
  - But only roughly: groups cases with the same outcome
- Condition coverage considers case analysis in more detail
  - Also *individual conditions* in a compound Boolean expression
    - E.g. both parts of `digit_high == 1 || digit_low == -1`

Basic Condition Testing

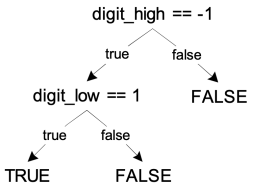
- Adequacy criterion: each basic condition must be executed at least once
- Coverage: no. of truth values taken by all basic conditions / (2 * total no. of basic conditions)

Basic conditions vs branches

- Basic condition adequacy criterion can be satisfied without satisfying branch coverage
- Branch and basic condition are not comparable (neither implies the other)

Covering branches and conditions

- Branch and condition adequacy
  - Coverall conditions and all decisions
- Compound condition adequacy
  - Cover all possible evaluations of compound conditions
  - Cover all branches of a decision tree



Compound conditions: exponential complexity

`(((a || b) && c) || d) && e`

Test Case	a	b	c	d	e
(1)	T	–	T	–	T
(2)	F	T	T	–	T
(3)	T	–	F	T	T
(4)	F	T	F	T	T
(5)	F	F	–	T	T
(6)	T	–	T	–	F
(7)	F	T	T	–	F
(8)	T	–	F	T	F
(9)	F	T	F	T	F
(10)	F	F	–	T	F
(11)	T	–	F	F	–
(12)	F	T	F	F	–
(13)	F	F	–	F	–

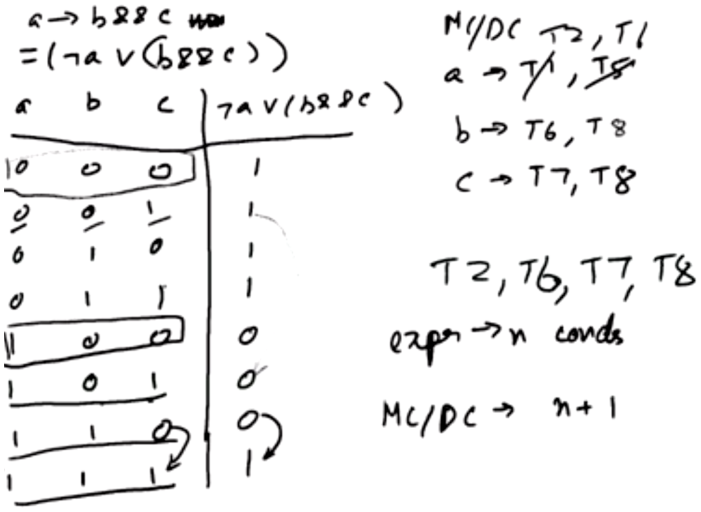
•short-circuit evaluation often reduces this to a more manageable number, but not always

Lecture 7 - Structural Testing: Part II

Modified condition/decision (MC/DC)

- Motivation: effectively test *important combinations* of conditions, without exponential blowup in test suite size
  - 'Important' combinations means: each basic condition shown to independently affect the outcome of each decision
- Requires:
  - For each basic condition C, two test cases
  - Values of all *evaluated* conditions except C are the same
  - Compound condition as a whole evaluates to *true* for one and *false* for the other
- "Every condition in my requirement should somehow affect the outcome, the reason that my requirement is true or false should be solely because of this condition and no other conditions."

Example from lecture



MC/DC: linear complexity

- N+1 test cases for N basic conditions

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

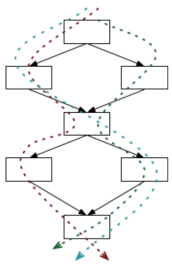
- Underlined values independently affect the output of the decision
- Required by the RTCA/DO-178B standard

Comments on MC/DC

- MC/DC is
  - Basic condition coverage (C)
  - Branch coverage (DC)
  - Plus one additional condition (M): every condition must *independently affect* the decision's output

- It is subsumed by compound conditions and subsumes all other criteria discussed so far
    - Stronger than statement and branch coverage
- A good balance of thoroughness and test size (and therefore widely used)

Paths (beyond individual branches)



- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
    - A pragmatic compromise will be needed
- A sequence of decisions were taken, and what the effect of that was
- There are dependencies imposed by a path
    - These dependencies can only be checked by multiple iterations before you can hit the faulty condition
    - E.g. buffer overflow

Path Adequacy

- Decision and condition adequacy criteria consider individual program decisions
- Path testing consider combinations of decisions along paths
- Adequacy criterion: each path must be executed at least once
- Coverage: no. of executed paths / total no.of paths

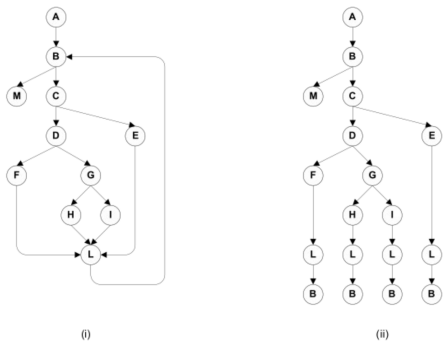
Practical path coverage criteria

- The number of paths in a program with loops is unbounded
    - Why? Because each iteration will result in different paths
    - The simple criterion is usually impossible to satisfy
- For a feasible criterion: partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
    - The number of traversals of loops (no. of iterations)
    - The length of the path to be traversed
    - The dependencies among selected paths

Boundary interior path testing

- Group together paths that differ only in the subpath they follow when repeating the body of a loop
    - Follow each path in the control flow graph up to the first repeated node
    - The set of paths from the root of the tree to each leaf is the required set of subpaths for boundary/interior coverage

Boundary interior adequacy for cgi-decode



Limits the number of iterations you take through your loop

Limitations of boundary interior adequacy

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- If you have if conditions within your loop, each of the if conditions will have different sub paths
- The number of paths can still grow exponentially (nested ifs)
- The subpaths through this control flow can include or exclude each of the statements Si, so that in total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be difficult, or even impossible if the conditions are not independent

Loop boundary adequacy

- Variant of the boundary interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: a test suite satisfies the loop boundary adequacy criterion iff for every loop:
    - In at least one test case, the loop body is iterated zero times
    - In at least one test case, the loop body is iterated once
    - In at least one test case, the loop body is iterated more than once
- Corresponds to the cases that would be considered a formal correctness proof for the loop

Cyclomatic adequacy

- Cyclomatic number: number of independent pats in the CFG
    - A path is representable as a bit vector, were each component of the vector represents an edge
    - ‘Dependence’ is ordinary linear dependence between (bit) vectors
- If e = no. of edges, n = no of nodes, c = no. of connected components or a graph, it is
    - Arbitrary graph = e - n + c
    - CFG = e - n + 2
- Cyclomatic coverage counts the number of independent paths that have been exercised, relative to cyclomatic complexity
- Not as rigorous, but helps check the number of minimal paths to test

Towards procedure call testing

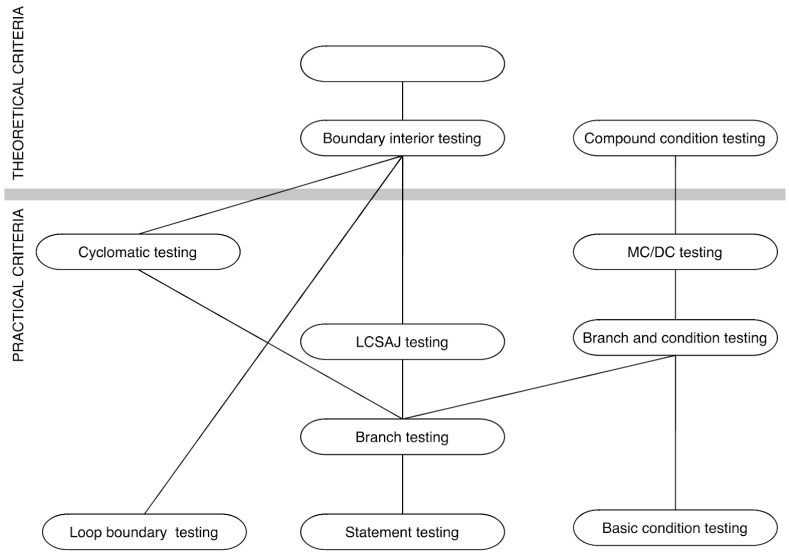
- Checks how the different procedures interact with each other
- The criteria considered to this point measure coverage of control flow of control flow within individual procedures
    - Not well suited to integration or system testing
- Choose a coverage granularity commensurate with the granularity of testing
    - If unit testing has been effective, then faults that remain to be found in integration testing will be primarily interface faults, and testing effort should focus on interfaces between units rather than their internal details

Procedure call testing

- Procedure entry and exit testing: procedure may have multiple entry points (e.g. Fortran) and multiple exit points
- Call coverage: the same entry point may be called from many points

Subsumption relation

If criteria A subsumes criteria B, it implies that if you achieve 100% of A you achieve 100% of B, but not the other way around. This also means that A is stronger than B.



Satisfying structural criteria

- Sometimes criteria may not be satisfiable
    - The criterion requires execution of
        - **Statements** that cannot be executed as a result of
            - Defensive programming
            - Code reuse (reusing code that is more general than strictly required for the application)
        - **Conditions** that cannot be satisfied as a result of
            - Interdependent conditions
        - **Paths** that cannot be executed as a result of
            - Interdependent decisions
- Large amounts of *fossil* code may indicate serious maintainability problems
    - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
    - Make allowances by setting a coverage goal less than 100%
    - Require justification of elements left uncovered
        - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC

Summary

- Adequacy criteria != test design techniques
- Different criteria address different classes of errors
- Full coverage is usually unattainable
    - Remember that attainability is an undecidable problem?
- When attainable, ‘inversion’ is usually hard
    - How do I find program inputs allowing me to cover something buried deeply in the CFG?
    - Automated support (e.g. symbolic execution) may be necessary
- Therefore, rater than requiring full adequacy, the ‘degree of adequacy’ of a test suite is estimated by coverage measures

Lecture 8 - Dependence and Data Flow Models

Why Data Flow Models

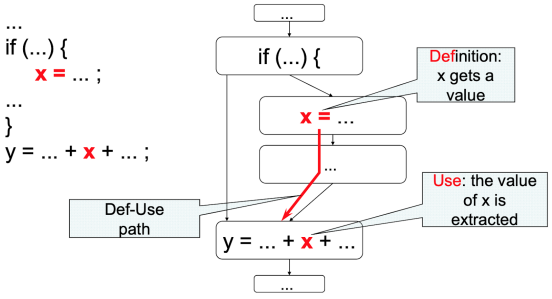
- Models from chapter 4 emphasis control
  - Control flow graph, call graph, finite state machines
- We also need to reason about dependence
  - Where does this value of x come from?
  - What would be affected by changing this?
- Many program analyses and test design techniques use data flow information
  - Often in combination with control flow
    - Example: 'taint' analysis to prevent SQL injection attacks
    - Example: dataflow test criteria (ch. 13)
- Security
  - Which parts of the program can be affected by user-defined variables
    - How do these variables propagate within your program
    - These parts of the program are considered vulnerable

Def-Use Pairs

- Associates a point in a program where a value is produced with a point where it is used
- **Definition:** where a variable gets a value
  - Variable declaration (often the special value 'uninitialised')
  - Variable initialisation
  - Assignment
  - Values received by a parameter
- **Use:** extraction of a value from a variable
  - Expressions
  - Conditional statements
  - Parameter passing
  - Returns

How do you pair them? (From CFG)

- If the definition can propagate to the use of the variable (there exists a path from definition to use)




```
/** Euclid's algorithm */
public class GCD
{
public int gcd(int x, int y) {
    int tmp;            // A: def x, y, tmp
    while (y != 0) {    // B: use y
       tmp = x % y;     // C: def tmp; use x, y
       x = y;           // D: def x; use y
       y = tmp;         // E: def y; use tmp
    }
    return x;           // F: use x
  }
```

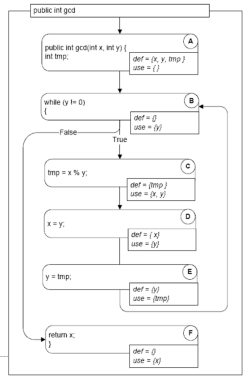
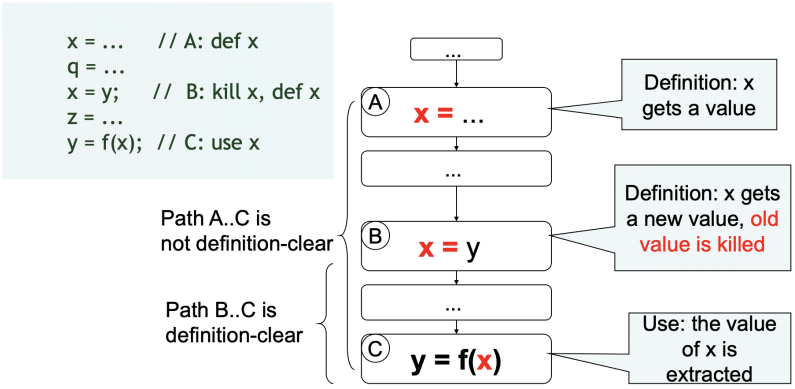


Figure 6.2, page 79

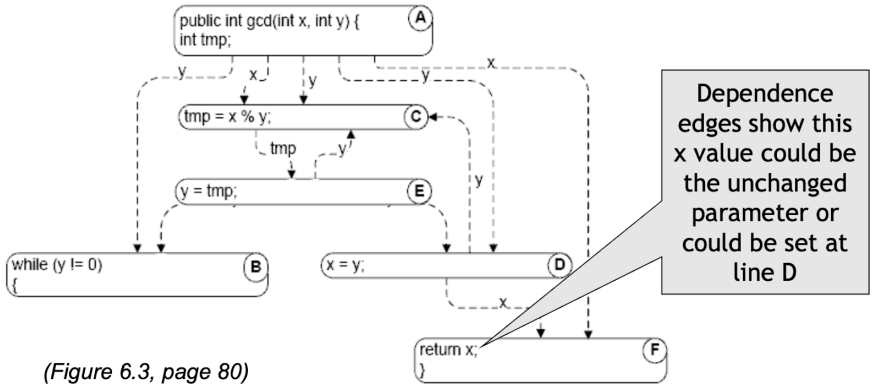
- A **definition-clear** path is a path along the CFG from a definition to a use of the same variable without another definition of the variable between
  - If, instead, another definition is present on the path, then the latter definition **kills** the former
- A def-use pair is formed if and only if there is a definition-clear path between the definition and the use

Definition-clear or Killing



(Direct) Data Dependence Graph

- A direct data dependence graph is
  - Nodes: as in the control flow graph (CFG)
  - Edges: def-use (DU) pairs, labeled with the variable name

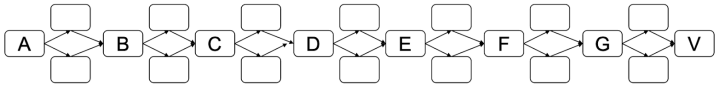


(Figure 6.3, page 80)

Calculating def-use pairs

- Definition-use pairs can be defined in terms of paths in the program control flow graph
  - There is an association (d, u) between a definition of variable v at d and a use of variable v at u iff
    - There is at least one control flow path from d to u
    - With no intervening definition of v
  - v_d **reaches** u (v_d is a **reaching definition** at u)
  - If a control flow path passes through another definition e of the same variable v, v_e **kills** v_d at that point
- Even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges
- Practical algorithms therefore do not search every individual path
  - Instead, they summarise the reaching definitions at a node over all the paths reaching that node
- Typical bugs caught by data flow testing: uninitialised variables, variables that are defined but are never used, taint analysis (check how variables propagate), security testing, data races, buffer overflows

Exponential paths (even without loops)



2 paths from A to B

4 from A to C

8 from A to D

16 from A to E

...

128 paths from A to V

Tracing each path is not efficient, and we can do much better.

Data Flow Algorithm

- An efficient algorithm for computing reaching definitions (and several other properties) is based on the way reaching definitions at onenode are related to the reaching definitions at an adjacent node
- Suppose we are calculating the reaching definitions of node *n*, and there is an edge (*p*, *n*) from an immediate predecessor node *p*
  - If the predecessor node *p* can assign a value to variable *v*, then the definition v_p reaches *n*
    - We say the definition v_p is generated at *p*
  - If a definition v_p of a variable *v* reaches a predecessor node *p*, and if *v* is not redefined at that node (in which case we say the v_p is killed at that point), then the definition is propagated on from *p* to *n*

Equations of node E (y = tmp)

Calculate reaching definitions at E in terms of its immediate predecessor D

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;            // A: def x, y, tmp
    while (y != 0) {    // B: use y
       tmp = x % y;     // C: def tmp; use x, y
       x = y;           // D: def x; use y
       y = tmp;         // E: def y; use tmp
    }
    return x;           // F: use x
  }
```

$Reach(E) = ReachOut(D)$

$ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$

Equations of node B (while (y != 0))

This line has two predecessors: Before the loop, end of the loop

```
public class GCD  {
public int gcd(int x, int y) {
    int tmp;            // A: def x, y, tmp
    while (y != 0) {    // B: use y
       tmp = x % y;     // C: def tmp; use x, y
       x = y;           // D: def x; use y
       y = tmp;         // E: def y; use tmp
    }
    return x;           // F: use x
  }
```

- $Reach(B) = ReachOut(A) \cup ReachOut(E)$
- $ReachOut(A) = gen(A) = \{x_A, y_A, tmp_A\}$
- $ReachOut(E) = (Reach(E) \setminus \{y_A\}) \cup \{y_E\}$

General equations for Reach analysis

Reach analysis will let you know where in your program code variable definitions can reach.

$$Reach(n) = \bigcup_{m \in pred(n)} ReachOut(m)$$

$$ReachOut(n) = (Reach(n) \setminus kill(n)) \cup gen(n)$$

$$gen(n) = \{ v_n \mid v \text{ is defined or modified at } n \}$$
$$kill(n) = \{ v_x \mid v \text{ is defined or modified at } x, x \neq n \}$$

Avail equations

- Used when you have a computed expression: can I reuse that expression or does it get modified?
- Forward analysis technique: checks what expressions are ready for use, if at any given point all the paths in CFG from entry to that point

$$Avail(n) = \bigcap_{m \in pred(n)} AvailOut(m)$$

$$AvailOut(n) = (Avail(n) \setminus kill(n)) \cup gen(n)$$

$$gen(n) = \{ exp \mid exp \text{ is computed at } n \}$$
$$kill(n) = \{ exp \mid exp \text{ has variables assigned at } n \}$$

Live variable equations

- Used often to check for unused variable definitions and gets rid of them
- Backward analysis: uses successors instead of predecessors

$$Live(n) = \bigcup_{m \in succ(n)} LiveOut(m)$$

$$LiveOut(n) = (Live(n) \setminus kill(n)) \cup gen(n)$$

$$gen(n) = \{ v \mid v \text{ is used at } n \}$$
$$kill(n) = \{ v \mid v \text{ is modified at } n \}$$

Classification of analyses

- forward/backward: a node's set depends on that of its predecessors/successors
- any-path/al-path: all node's set contains a value if and only if it is coming from any/all of its inputs

	Any-path (union)	All-paths (intersection)
Forward (predecessors)	Reach	Avail
Backward (successors)	Live	"Inevitable"

Iterative Solution of Dataflow Equations

- Initialise values (first estimate of answer)
    - For ‘any-path’ problems, first guess is ‘nothing’ (empty set) at each node
    - For ‘all-paths’ problems, first guess is ‘everything (set of all possible values = union of all ‘gen’ sets)
- Repeat until nothing changes
    - Pick some node and recalculate (new estimate)
- This will converge on a ‘fixed point’ solution where every new calculation produces the same value as the previous guess

Cooking your own: From Execution to Conservative Flow Analysis

- We can use the same data flow algorithms to approximate other dynamic properties
    - Gen set will be ‘facts that become true here’
    - Kill set will be ‘facts that are no longer true here’
    - Flow equations will describe propagation
- Example”: taintedness (in web form processing)
    - ‘Taint’: a user-supplied value (e.g. from web form) that has not been validated
    - Gen: we get this value from an untrusted source here
    - Kill: we validated to make sure the value is proper

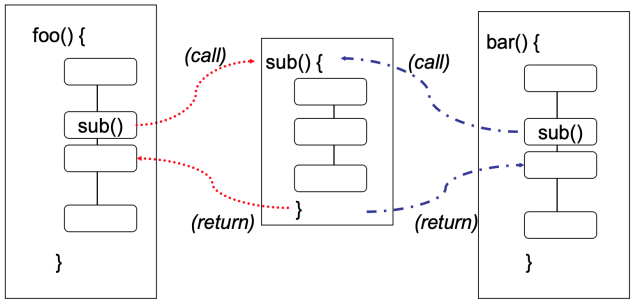
Data flow analysis with arrays and pointers

- Arrays and pointers introduce uncertainty: you don’t know if two different variables point to the same memory location. Do different expressions access the same storage?
    - a[i] same as a[k] when i = k
    - a[i] same as b[i] when a = b (aliasing)
- The uncertainty is accommodated depending to the kind of analysis
    - Any-path: gen sets should include all potential aliases and kill set should include only what is definitely modified
    - All-path: kill sets should include all potential aliases and gen set should include only what is definitely modified (vice versa)

Scope of Data Flow Analysis

- Intraprocedural: within a single method or procedure
- Interprocedural: across several methods (and classes) or procedures
- cost/precision trade-offs for interprocedural analysis are critical, and difficult
    - Context sensitivity
    - Flow sensitivity

Context Sensitivity



A **context-sensitive** (interprocedural) analysis distinguishes sub() called from foo() from sub() called from bar();
A **context-insensitive** (interprocedural) analysis does not separate them, as if foo() could call sub() and sub() could then return to bar()

Flow Sensitivity

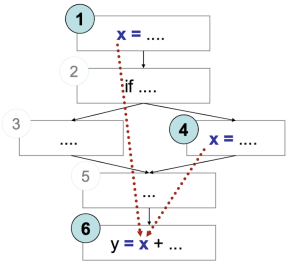
- Not about connections within the methods, but about the order of statements within your program
- Reach, Avail, etc. were **flow-sensitive**, **intraprocedural** analysis
    - They considered ordering and control flow decisions
    - Within a single procedure or method, this is (fairly) cheap - $O(n^3)$ for n CFG nodes
- Many **interprocedural** flow analyses are **flow-insensitive**
    - $O(n^3)$ would not be acceptable for all the statements in a program!
        - Though $O(n^3)$ on each individual procedure might be okay
    - Often flow-insensitive analysis is not good enough - consider type-checking as an example

Lecture 9 - Data Flow Testing

Motivation

- Middle ground in structural testing
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases
    - And only a few paths uncover additional faults anyway
  - Need to distinguish 'important' paths
- Intuition: statements interact through data flow
  - Value computed in one statement, used in another
  - Bad value computation revealed only when it is used

Data flow concept

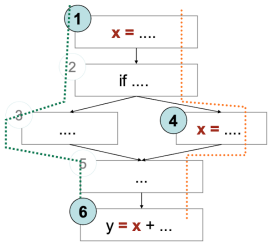


- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are ***def-use (DU) pairs***
  - defs at 1,4
  - use at 6

Terms

- **DU pair:** a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use
  - x = … is a *definition* of x
  - foo(x) is a *use* of x
- **DU path:** a definition-clear path on the CFG starting from a definition to a use of a same variable
  - Definition clear: value is not replaced on path
  - Note- loops could create infinite DU paths between a definition and a use

Definition-clear path



- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4
- (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

Adequacy criteria

- All DU pairs: each DU pair is exercised by at least one test case
- All DU paths: each *simple* (non-looping) DU path is exercised by at least one test case
- All definitions: for each definition, there is at least one test case which exercises a DU pair containing it
  - Every computed value is used somewhere
- Corresponding coverage fractions can also be defined

Difficult cases

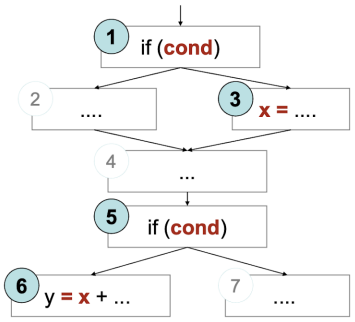
- x[i] = ... ; ... ; y = x[j]
  - DU pair (only) if i==j
- p = &x ; ... ; *p = 99 ; ... ; q = x
  - *p is an alias of x
- m.putFoo(...); ... ; y=n.getFoo(...);
  - Are m and n the same object?
  - Do m and n share a “foo” field?

- Problem of ***aliases***: Which references are (always or sometimes) the same?

Data flow coverage with complex structures

- Arrays and pointers are critical for data flow analysis
  - Under-estimation of aliases may fail to include some DU pairs
  - Over-estimation, on the other hand, may introduce unfeasible test obligations
- For testing, it may be preferable to accept underestimate of alias set rather than over-estimation or expensive analysis
  - Controversial: in other applications (e.g. compilers), a conservative over-estimation of aliases is usually required
  - Alias analysis may rely on external guidance or other global analysis to calculate good estimates
  - Undisciplined use of dynamic storage, pointer arithmetic, etc. may make the whole analysis infeasible

Infeasibility



- Suppose ***cond*** has not changed between 1 and 5
  - Or the conditions could be different, but the first implies the second
- Then (3,5) is not a (feasible) DU pair
  - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
  - No test case can cover them

- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
  - Combinations of elements matter!
  - Impossible to (infallibly) distinguish feasible from infeasible paths
    - More paths = more work to check manually
- In practice, reasonable coverage is (often, not always) achievable
  - Number of paths is exponential in worst case but number of paths is exponential in worst case, but often linear
  - All DU paths is more often impractical

Lecture 10 - Test Case Selection and Adequacy Criteria

Adequacy: we can't get what we want

- What we would like: a real way of measuring effective testing
    - If the system passes an adequate suite of test cases, then it must be correct (or dependable)
- But that's impossible!
    - Adequacy test suites, in the sense above, is provably undecidable
- So we'll have to settle on weaker proxies for adequacy
    - Design rules to highlight inadequacy of test suites

Practical (in)adequacy criteria

- Criteria that identify inadequacies in test suites. Examples:
    - If the specification describes different treatment in two cases, but the test suite does not check that the two cases are in fact treated differently, we may conclude that the test suite is inadequate to guard against faults in the program logic
    - If no test in the test suite executes a particular program statement, the test suite is inadequate to guard against faults in that statement
- If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite
- If a test suite satisfies all the obligations by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness

Terminology

- **Test case:** a set of inputs, execution conditions, and a pass/fail criterion
- **Test case specification:** a requirement to be satisfied by one or more test cases
- **Test obligation:** a partial test case specification, requiring some property deemed important to thorough testing
- **Test suite:** a set of test cases
- **Test or test execution:** the activity of executing test cases and evaluating their results
- **Adequacy criterion:** a predicate that is true (satisfied) or false (not satisfied) of a <program, test suite> pair

Where do test obligations come from?

- **Functional (black box, specification-based):** from software specifications
    - Example: if specification requires robust recovery from power failure, test obligations should include simulated power failure
- **Structural (white or glass box):** from code
    - Example: traverse each program loop one or more times
- **Model-based:** from model of system
    - Models used in specification or design, or derived from code
    - Example: exercise all transitions in communication protocol model
- **Fault-based:** from hypothesised faults (common bugs)
    - Example: check for buffer overflow handling (common vulnerability) by testing on very large inputs

Adequacy criteria

- Adequacy criterion = set of test obligations
- A test suite satisfies an adequacy criterion if
  - All tests succeed (pass)
  - Every test obligation in the criterion is satisfied by at least one of the test cases in the test suite
  - Example:
    - The statement coverage adequacy criterion is satisfied by test suite S from program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”

Satisfiability

- Sometimes *no* test suite can satisfy a criterion for a given program
  - Example: defensive programming style includes ‘can’t happen’ sanity checks

```
if (z < 0) {
      throw new LogicError("z must be positive here!");
}
```

  - No test suite can satisfy statement coverage for this program (if it’s correct)

Coping with unsatisfiability

- Approach A: exclude any unsatisfiable obligation from the criterion
  - Example: modify statement coverage to require execution only of statements that can be executed
  - But we can’t know for sure which are executable
- Approach B: measure the extent to which a test suite approaches an adequacy criterion
  - Example: if a test suite satisfies 85 out of 100 obligations, we have reached 85% coverage
    - Terms: an adequacy criterion is satisfied or not, a coverage measure is the fraction of satisfied obligations

Comparing Criteria

- Can we distinguish stronger from weaker adequacy criteria?
- Empirical approach: study the effectiveness of different approaches to testing in industrial practice
  - What we really care about, but…
  - Depends on the setting; may not generalise from one organisation or project to another
- Analytical approach: describe conditions under which one adequacy criterion is provable stronger than another
  - Stronger = gives stronger guarantees
  - One piece of the overall ‘effectiveness’ question

The *subsumes* approach

Test adequacy criterion A subsumes test adequacy criterion B, if and only of for every program P, every test suite satisfying A with respect to P also satisfies B with respect to B.

- Example: exercising all program branches (branch coverage) *subsumes* exercising all program statements
- A common analytical comparison of closely related criteria
  - Useful for working from easier to harder levels of coverage, but not a direct indication of quality

Uses of Adequacy Criteria

- Test selection approaches

    - Guidance in devising thorough test suite
        - Example: a specification-based criterion may suggest test cases covering representative combinations of values
- Revealing missing tests
    - Post hoc analysis: what do I might have missed with this test suite?
- Often in combination
    - Example: design test suite from specifications, then use structural criterion (e.g. coverage of all branches) to highlight missed logic

Summary

- Adequacy criteria provide a way to define a notion of ‘thoroughness’ in a test suite
    - But they don’t offer guarantees; more like design rules to highlight inadequacy
- Defined in terms of ‘covering’ some information
    - Derived from many sources: specifications, code, models
- May be used for selection as well as measurement
    - With caution! An aid to thoughtful test design, not a substitute

Lecture 11 - Mutation Testing

Definitions

- Fault-based testing: directed towards 'typical' faults that could occur in a program
- Basic idea:
  - Take a program and test data generated for that program
  - Create a number of similar programs (mutants), each differing from the original in one small way (i.e. each possessing a fault) e.g. replace addition operator by multiplication operator
  - The original test data are then run through the mutants
  - If the test data detects all differences in mutants, then the mutants are said to be dead, and the test set is adequate

Different types of mutants

- **Stillborn mutants:** syntactically incorrect, killed by compiler, e.g. x = a ++ b
- **Trivial mutants:** killed by almost any test case
- **Equivalent mutant:** acts in the same behavior as the original program, e.g. x = a + b and x a - (-b)
- None for the above are interesting from a mutation testing perspective
- Those mutants which are interesting are the ones that behave differently than the original program, and we do not have test cases to identify them (to cover those specific changes)

Example of an equivalent mutant

Original program

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index==10)
        break;
}
```

A mutant

```
int index=0;
while (...)
{
    . . .;
    index++;
    if (index>=10)
        break;
}
```

The mutant above will always break once it reaches 10, so it doesn't matter if the value is == 10 or > 10.

Basic Ideas

In Mutation Testing:

1. We take a program and a test suite generated for that program (using other test techniques)
2. We create a number of *similar* programs (mutants), each differing from the original in one small way, i.e. each possessing a fault
   a. E.g. replacing an addition operator by a multiplication operator
3. The original test data are then run on the *mutants*
4. If test cases detect differences in mutants, then the mutants are said to be dead (*killed*), and the test set is considered *adequate*

Furthermore:

- A mutant remains *live* either
  - Because it is equivalent to the original program (functionally identical although syntactically different - called an *equivalent mutant*), or
  - The test set is inadequate to kill the mutant
    - If this is the case, then the test data need to be augmented (by adding one or more new test cases) to kill the *live* mutant
- For the automated generation of mutants, we use *mutation operators*, that is predefined program modification rules (i.e. corresponding to a fault model)

Example of Mutation Operators

- Constant replacement
- Scalar variable replacement
- Scalar variable for constant replacement
- Constant for scalar variable replacement
- Array reference for constant replacement
- Array reference for scalar variable replacement
- Constant for array reference replacement
- Scalar variable for array reference replacement
- Array reference for array reference replacement
- Source constant replacement
- Data statement alteration
- Comparable array name replacement
- Arithmetic operator replacement
- Relational operator replacement
- Logical connector replacement
- Absolute value insertion
- Unary operator insertion
- Statement deletion
- Return statement replacement

Example of Mutation Operators specific to object-oriented programming languages:

- Replacing a type with a compatible subtype (inheritance)
- Changing the access modifier of an attribute, a method
- Changing the instance creation expression (inheritance)
- Changing the order of parameters in the definition of a method
- Changing the order of parameters in a call
- Removing an overloading method
- Reducing the number of parameters
- Removing an overriding method
- Removing a hiding Field
- Adding a hiding field

Specifying Mutations Operators

- Ideally, we would like the mutation operators to be representative of (and generate) all realistic types of faults that could occur in practice
- Mutation operators change with programming languages, design and specification paradigms, though there is much overlap
- In general, the number of mutation operators is large as they are supposed to capture all possible syntactic variations in a program
- Recent paper suggests random sampling of mutants can be used
- Some recent studies seem to suggest that mutants are good indicators of test effectiveness (Andrews et al, ICSE 2005)

Mutation Coverage

- Complete coverage equals to killing all non-equivalent mutants (or random sample)
- The amount of coverage is also called “mutation score”
- We can see each mutant as a test requirement
- The number of mutants depends on the definition of mutation operators and the syntax/structure of the software
- Numbers of mutants tend to be large, even for small programs (hence random sampling)

A simple example

Original Function		With Embedded Mutants
`int Min (int A, int B)`		`int Min (int A, int B)`
`int minVal;`		`int minVal;`
`{`		`{`
`minVal = A;`		`minVal = A;`
`if (B < A)`	Δ1	`minVal = B;`
`{`		`if (B < A)`
`minVal = B;`	Δ2	`if (B > A)`
`}`	Δ3	`if (B < minVal)`
		`{`
		`minVal = B;`
	Δ4	`Bomb();`
	Δ5	`minVal = A;`
	Δ6	`minVal = failOnZero (B);`
		`}`
`return (minVal);`		`return (minVal);`
`} // end Min`		`} // end Min`

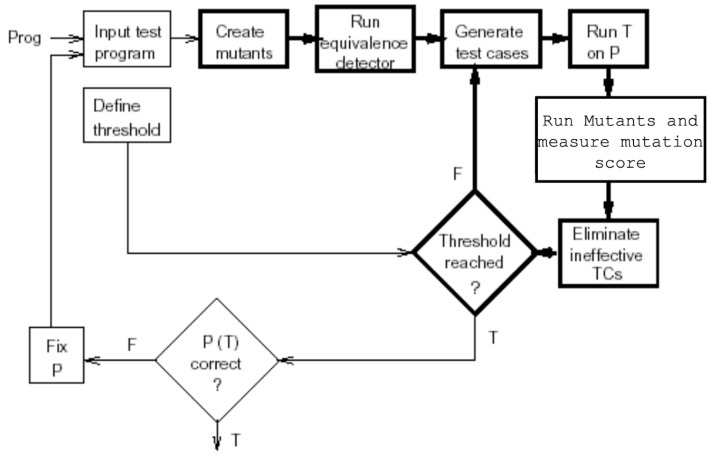
Deltas represent syntactic modifications. In fact, each of them will be embedded in a different program version, a mutant.

- Mutant 3 is equivalent as, at this point, minVal and A have the same value
- Mutant 1: In order to find an appropriate test case to kill it, we must
  - Reach the fault seeded during execution (Reachability)
    - Always true (i.e., we can always reach the seeded fault)
  - Cause the program state to be incorrect (Infection)
    - A <> B
  - Cause the program output and/or behavior to be Incorrect (Propagation)
    - (B < A) = false

Assumptions

- What about more complex errors, involving several statements?
- Let's discuss two assumptions:
  1. Competent programmer assumption: they write programs that are nearly correct
  2. Coupling effect assumption: test cases that distinguish all programs differing from a correct one by only simple errors is so sensitive that they also implicitly distinguish more complex errors
- There is some empirical evidence of the above two hypotheses: Offutt, A.J., Investigations of the Software Testing Coupling Effect, ACM Transactions on Software Engineering and Methodology, vol. 1 (1), pp. 3-18, 1992

Mutation Testing Process



Jeff Offutt, A Practical System for Mutation Testing: Help for the Common Programmer

Summary

- It measures the quality of test cases
- A tool's slogan: "Jester - the JUnit test tester"
- It provides the tester with a clear target (mutants to kill)
- Mutation testing can also show that certain kinds of faults are unlikely (those specified by the fault model), since the corresponding test case will not fail

- It does force the programmer to inspect the code and think of the test data that will expose certain kinds of faults
- It is computationally intensive, a possibly very large number of mutants is generated: random sampling, selective mutation operators (Offutt)
- Equivalent mutants are a practical problem: It is in general an undecidable problem
- Probably most useful at unit testing level

Other Applications

- Mutation operators and systems are also very useful for assessing the effectiveness of test strategies – they have been used in a number of case studies
    - Define a set of realistic mutation operators
    - Generate mutants (automatically)
    - Generate test cases according to alternative strategies
    - Assess the mutation score (percentage of mutants killed)
- In our discussion, we saw mutation operators for source code (body)
- There are also works on:
    - Mutation operators for module interfaces (aimed at integration testing)
    - Mutation operators on specifications: Petri-nets, state machines, … (aimed at system testing)

Lecture 12 - Test-Driven Development

What is Test-Driven Development (TDD)?

- Also known as test-driven design
- A methodology
- Common TDD misconception
    - TDD is not about testing
    - TDD is about design and development
    - By testing first you design your code

Things that characterise TDD

- Short development iterations (short release cycles, 4-6 weeks)
- Based on requirement and pre-written test cases
- Produces code necessary to pass that iteration's test
- Refactor both code and tests
- The goal is to produce working clean code that fulfills requirement

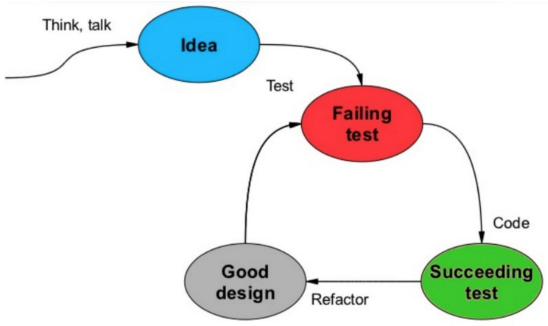
Principle of TDD

- Kent Beck defines:
    - Never write a single line of code unless you have a failing automated test
    - Eliminate duplication
- Steps:
    - Red: automated test fail
    - Green: automated test pass because dev code has been written
    - Refactor: eliminate duplication, clean the code

TDD Basics: Unit Testing

- Make it fail: no code without a failing test
- Make it work: as simply as possible
- Make it better: refactor

How does TDD help?

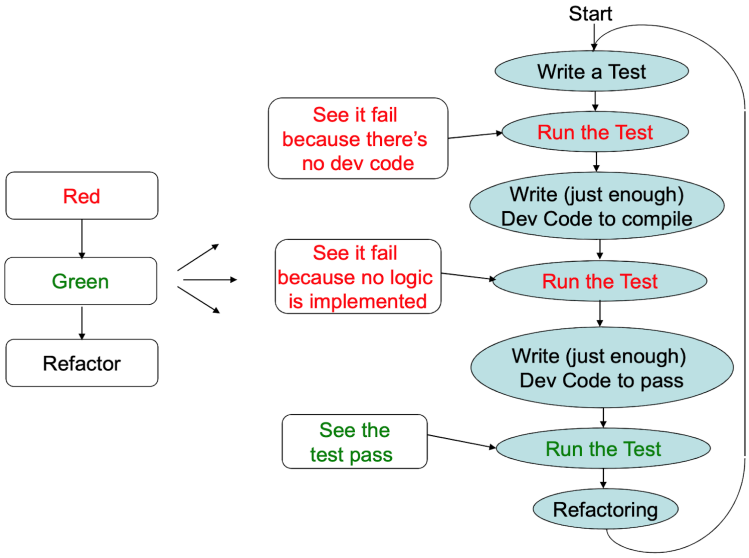


TDD Cycle

- Write test code
    - Guarantees that every functional code is testable
    - Provides a specification for the functional code
    - Helps to think about design
    - Ensure the functional code is tangible

- Write functional code
    - Fulfill the requirement (test code)
    - Write the simplest solution that works
    - Leave improvements for a later step
    - The code written is only designed to pass the test
        - No further (and therefore untested code is not created)
- Refactor
    - Clean-up the code (test and functional)
    - Make sure the code expresses intent
    - Remove code smells
    - Re-think the design
    - Delete unnecessary code

Principle of TDD (In Practice)



Why do we do TDD?

- Confidence in change
    - Increase confidence in code
    - Fearlessly change your code
- Document requirements
- Discover usability issues early
- Regression testing = stable software = quality software

Advantages of TDD

- TDD Shortens the programming feedback loop
- TDD Promotes the development of high-quality code (code is always tested)
- User requirements more easily understood (you are forced to understand the requirements earlier on)
- Reduced interface misunderstandings
- TDD provides concrete evidence that your software works
- Reduced software defect rates
- Better code
- Less debug time

Disadvantages of TDD

- Programmers like to code, not to test
- Test writing is time consuming
- Test completeness is difficult to judge
- TDD may not always work

There are other kinds of tests

- Unit test (unit)
- Integration test (collaboration)
- User interface test (frontend)
- Regression test (continuous integration)
- System, performance, stress, usability, etc.
- White-box testing
- Black-box testing

The only tests relevant to TDD is black-box unit testing!

How to do TDD

- Just look at the component you want to perform TDD on
    - You don't have to worry about the component's interactions with the other components
- Do not write the code in your head before you write the test
- When you first start at doing TDD you 'know' what the design should be
    - You 'know' what the code you want to write looks like
    - So you write a test that will let you write that bit of code
- When you do this you aren't really doing TDD - since you are writing the code first (even if the code is only in your head)
- It takes some time to realise that you need to focus on the test
- Write the test for the behavior you want, then write the minimal code needed to make it pas
- Let the design emerge through refactoring and repeat until done

Unbounded Stack Example

Requirement: FILO/LIFO messaging system

Brainstorm a list of tests for the requirement:

- Create a *Stack* and verify that *IsEmpty* is true
- *Push* a single object on the *Stack* and verify that *IsEmpty* is false
- *Push* a single object, *Pop* the object, and verify that *IsEmpty* is true
- *Push* a single object, remembering what it is; *Pop* the object, and verify that the two objects are equal
- *Push* three objects, remembering what they are; *Pop* each one, and verify that they are removed in the correct order
- *Pop* a *Stack* that has no elements
- *Push* a single object and then call *Top*
- Verify that *IsEmpty* is false
- *Push* a single object, remembering what it is; and then call *Top*
- Verify that the object that is returned is the same as the one that was pushed
- Call *Top* on a Stack with no elements.

Choosing the first test

- The simplest: if you need to write code that is untested, choose a simpler test
- The essence: if the essence approach takes too much time to implement, choose a simpler test

Anticipating future tests

- In the beginning, focus on the test you are writing, and do not think of the other tests
- As you become familiar with the technique and the task, you can increase the size of the steps
- But remember still, no written code must be untested

TDD by examples

- Write a method that reverse last 2 characters of a string
    - If null -> return null
    - If empty -> return empty
    - If length of string equal 1 -> return itself
    - Example inputs and outputs
        - “A” -> “A”
        - “” -> “”
        - Null -> null
        - “AB” -> “BA”
        - “RAIN” -> “RANI”
        - “AA” -> “AA”
    - Create first test case
    - Cannot compile because StringHelper class is not created
    - After StringHelper class is created, run ALL TEST CASES to see they (or one of them) fail
    - Make a little change to pass these test cases
    - Refactor code to pass the test cases
- For more detailed diagrams, open this link

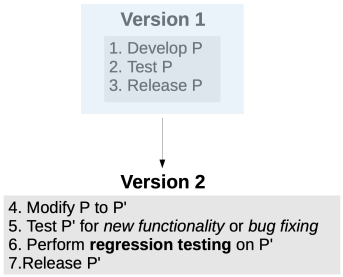
Lecture 13 - Regression Testing

Evolving Software

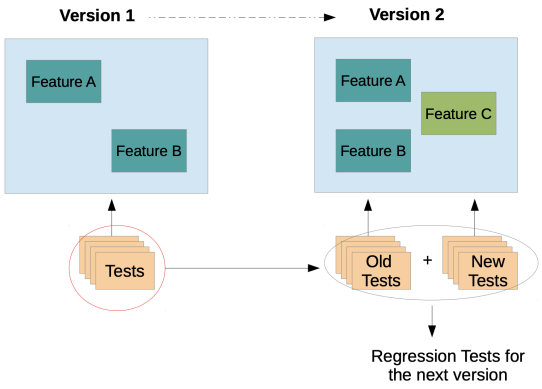
- Large software systems are usually built incrementally
  - Maintenance - fixing errors and flaws, hardware changes
  - Enhancements - new functionality, improved efficiency, extension, new regulations

Regressions

- Ideally, software should *improve* over time
- But changes can both
  - Improve software, adding features and fixing bugs
  - Break software, introducing new bugs
- We can call such breaking changes regressions
- Make sure that the tests that were passing before still passed and the new tests also pass



Example



Consequences of Poor Regression Testing

- Thousands of 1-800 numbers disabled by a poorly tested software upgrade (December 1991)
- Fault in an SS& software patch causes extensive phone outages (June 1991)
- Fault in a 4ESS upgrade causes massive breakdown in the AT&T network (January 1990)

```
1  While (ring receive buffer | empty and side buffer | empty)
2  {
3  Initialize pointer to first message in side buffer or ring received buffer
4  Get a copy of buffer
5  Switch (message) {
6  Case incoming message: if (sending switch = out of service)
7        {
8        if (ring write buffer = empty)
9        Send in service to states map manager;
10       Else
11       Break;
12       }
13 Process incoming message, set up pointers to optional parameters
14       Break;
15       :
16       }
17 Do optional parameter work
18 }
```

Regression

- Yesterday it worked, today it doesn't
    - I was fixing X and accidentally broke Y
- Tests must be re-run after any change
    - Adding new features
    - Changing, adapting software to new conditions
    - Fixing other bugs
- Regression testing can be a major cost of software maintenance
    - Sometimes much more than making the change
- Takes too long
    - Regression testing is proportional to the size of the product, rather than size of the change

Basic problems of regression test

- Maintaining test suite
    - If I change feature X, how many test cases must be revised because they use feature X?
    - Which test cases should be removed or replaced? Which test cases should be added?
- Cost of re-testing
    - Often proportional to product size, not change size
    - Big problem if testing requires manual effort
    - Possible problem even for automated testing, when the test suite and test execution time grows beyond a few hours

Test Case Maintenance

- Some maintenance is inevitable
    - If feature X has changed, test cases for feature X will require updating
- Some maintenance could be avoided
    - Example: trivial changes to user interface or file format should not invalidate large numbers of test cases
- Test suites should be modular!
    - Avoid unnecessary dependance
- Generating concrete test cases from test case specifications can help

Obsolete and Redundant

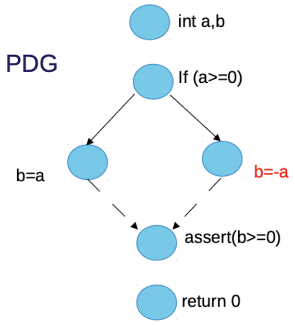
- Obsolete: a test case that is no longer valid
    - Should be removed from the test suite
- Redundant: a test case that does not differ significantly from others
    - Unlikely to find a fault missed by similar test cases
    - Has some cost in re-execution
    - May or may not be removed, depending on costs

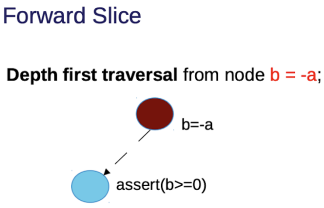
Regression Test Optimisation

- **Re-test all**
    - Traditional approach - select all (too expensive!)
    - The test-all approach is good when you want to be certain that the new version works on all tests developed for the previous version
    - What if you only have limited resources to run tests and have to meet a deadline?
    - Those on which the new and the old program produce different outputs (undecidable)
    - No test prioritisation

- **Regression test selection**
  - From the entire test suite, only select subset of test cases whose execution is relevant to changes
  - Code-based regression test selection
    - Observation: a test case can't find a fault in code it doesn't execute
      - In a large system, many parts of the code are untouched by many test cases
    - So, only execute test cases that execute changed or new code
  - Control-flow and Data-flow regression test selection
    - Same basic idea as code-based selection
      - Re-run test cases only if they include changed elements
      - Elements may be modified control flow nodes and edges, or definition-use (DU) pairs in data flow
    - To automate selection
      - Tools record elements touched by each test case
        - Stored in database of regression test cases
      - Tools note changes in program
      - Check test-case database for overlap
  - Specification-based regression test selection
    - Like code-based and structural regression test case selection
      - Pick test cases that test new and changed functionality
    - Difference: no guarantee of independence
      - A test case that isn't 'for' changed or added feature X might find a bug in feature X anyway
    - Typical approach: specification-based prioritisation
      - Execute all test cases, but start with those that related to changed and added feature
  - Example

```
int main()
{
        int a, b;
        if (a>=0)
                b = a;
        else
                b = -a;
        assert(b >= 0);
        return 0;
}
```



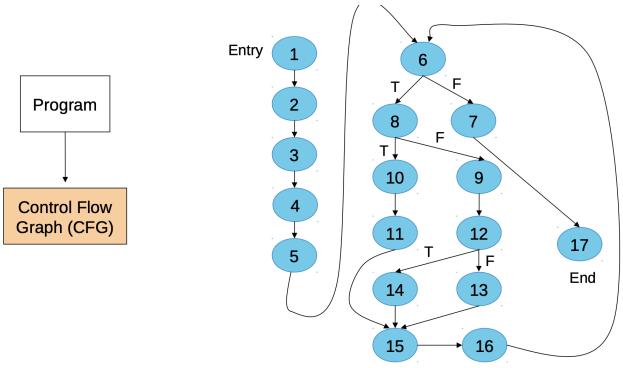


  - Slicing procedure

Computing the greatest in an array of integers

```
int main(int argc, char* argv[]) {
  unsigned int num[5] = {12, 23, 4, 78, 34};
  unsigned int largest, counter = 0;
  while (counter < 5) {
          if (counter ==0)
                  largest = num[counter];
          else if(largest > num[counter])
                  largest = num[counter];
          ++counter;
   }
  for (counter = 0; counter < 5; counter++)
          assert(largest >= num[counter];
}
```

- Construct Control Flow Graph



- Build a Program Dependence Graph (PDG) that captures control and data dependencies between nodes in CFG
- Sample Data Dependency

For *counter* variable

1 → 2,3,4,5,6,7

7 → 2,3,4,5,6,7

```
int main(int argc, char* argv[]) {
 1  unsigned int num[5] = {12, 23, 4, 78, 34},
largest,   counter = 0;
 2  while (counter <5) {
 3         if (counter ==0)
 4                  largest = num[counter];
 5         else if(largest < num[counter])
 6                  largest = num[counter];
 7         counter = counter +1;
       }
     }
```

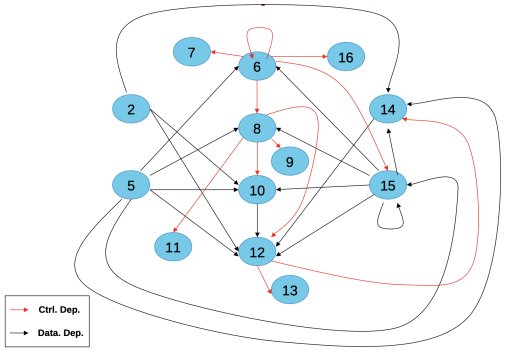
- Sample Control Dependency

Conditional in statement 3

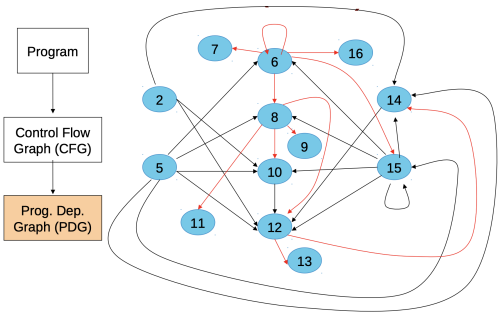
3 → 4, 5

```
int main(int argc, char* argv[]) {
 1  unsigned int num[5] = {12, 23, 4, 78, 34},
largest,   counter = 0;
 2  while (counter <5) {
 3         if (counter ==0)
 4                  largest = num[counter];
 5         else if(largest < num[counter])
 6                  largest = num[counter];
 7         counter = counter +1;
       }
     }
```

- PDG for Example Program



- Slicing procedure (so far)



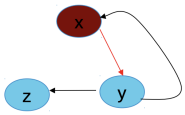
- Slight change in the example

```
int main(int argc, char* argv[]) {
  unsigned int num[5] = {12, 23, 4, 78, 34},
           largest,   counter = 0;
  while (counter <5) {
           if (counter ==0)
                    largest = num[counter];
           else if(largest > num[counter])
                    largest = num[counter];
           ++counter;
  }
  for (counter = 0; counter < 5;
                    counter++)
           assert(largest >= num[counter];
}
```

Changed program

- Forward Slicing from Changes
  - Compute the nodes corresponding to changed statements in the PDG, and
  - Compute a transitive closure over all forward dependencies (control + data) from these nodes
- Forward slice
  - Only the three statements that are affected, so you keep the tests that exercise these three statements and discount the remaining

Depth first traversal from changed node



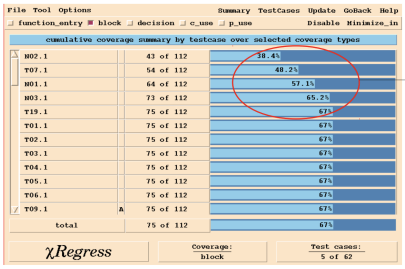

```
else if(largest < num[counter])
         largest = num[counter];
assert (largest >= num[counter];
```

- **Regression test set minimisation**
  - Identify test cases that are redundant and remove them from the test suite to reduce its size
  - Most simple heuristic: branch coverage
  - Test set attributes: coverage, size, effectiveness (fault finding; maximise this!)
  - Structural coverage
    - (In)adequacy criteria
      - If significant parts of the program structure are not tested, testing is surely inadequate
    - Control flow coverage criteria
      - Statement (node, basic block) coverage
      - Branch (edge) coverage
      - Condition coverage
      - Path coverage
      - Data flow (syntactic dependency) coverage
    - Attempted compromise between the impossible and the inadequate
  - Test set minimisation
    - Maximise coverage with minimum number of test cases
    - The minimisation algorithm can be exponential in time
    - Does not occur in our experience
      - Some examples
        - An object-oriented language compiler (100 KLOC)
        - A provisioning application (353 KLOC) with 32K regression tests
        - A database application with 50 files (35 KLOC)
        - A space application (10 KLOC)

- Stop after a pre-defined number of iterations
- Obtain an approximate solution by using a greedy heuristic

- Example

Sort test cases in order of increasing cost per additional coverage



Only 5 of the 62 test cases are included in the minimized subset which has the same block coverage as the original test set.

- **Regression test set prioritisation**
  - Sort test cases in order of increasing cost per additional coverage
  - Select the first test case
  - Repeat the above two steps until n test cases are selected or max cost is reached (whichever is first)
  - Example
    - Individual decision coverage and cost per test case

```
$ atac -K -md main.atac wc.atac wordcount.trace

cost   % decisions   test
------ ------------ ------------
120    57(20/35)    wordcount.1
50     11(4/35)     wordcount.2
20     49(17/35)    wordcount.3
10     11(4/35)     wordcount.4
40     71(25/35)    wordcount.5
60     60(21/35)    wordcount.6
80     11(4/35)     wordcount.7
20     66(23/35)    wordcount.8
10     66(23/35)    wordcount.9
70     60(21/35)    wordcount.10
50     60(21/35)    wordcount.11
50     60(21/35)    wordcount.12
50     20(7/35)     wordcount.13
40     14(5/35)     wordcount.14
60     60(21/35)    wordcount.15
20     26(9/35)     wordcount.16
150    54(19/35)    wordcount.17
900    100(35)      == all ==
```

    - Prioritised cumulative decision coverage and cost per test case

```
$ atac -Q -md main.atac wc.atac wordcount.trace

cost   % decisions   test
(cum)  (cumulative)
------ ------------ ------------
10     66(23/35)    wordcount.9
30     77(27/35)    wordcount.3
40     83(29/35)    wordcount.4
60     89(31/35)    wordcount.8
100    91(32/35)    wordcount.5
140    94(33/35)    wordcount.14
200    97(34/35)    wordcount.15
280    100(35)      wordcount.7
300    100(35)      wordcount.16
350    100(35)      wordcount.2
400    100(35)      wordcount.12
450    100(35)      wordcount.11
500    100(35)      wordcount.13
560    100(35)      wordcount.6
630    100(35)      wordcount.10
750    100(35)      wordcount.1
900    100(35)      wordcount.17
```

Cost per additional coverage
10/23 = 0.43
(30-10)/(27-23) = 20/4 = 5.00
(40-30)/(29-27) = 10/2 = 5.00
(60-40)/(31-29) = 20/2 = 10.00
(100-60)/(32-31) = 40/1 = 40.00

Increasing Order

- Prioritised rotating selection
  - Basic idea:
    - Execute all test cases, eventually
    - Execute some sooner than others
  - Possible priority schemes:
    - Round robin: priority to least-recently-run test cases
    - Track record: priority to test cases that have detected faults before
      - They probably execute code with a high fault density
    - Structural: priority for executing elements that have not been recently executed
      - Can be coarse-grained: features, methods, files, etc.

Lecture 13 - Security Testing

Security Testing vs 'Regular' Testing

- 'Regular' testing aims to ensure that the program meets customer requirements in terms of feature and functionality
- Tests 'normal' use cases
    - Test with regards to common expected usage patterns
- Security testing aims to ensure that program fulfills security requirements
    - Often non-functional
    - More interested in misuse cases; focuses more on the invalid space rather than the valid space; attackers taking advantage of 'weird' corner cases

Functional vs non-functional security requirements

- Functional requirements - *what* shall the software do?
- Non-functional requirements - *how* should it be done?
- Regular functional requirement example (webmail system): it should be possible to use HTML formatting in emails
- Functional security requirement example: the system should check user registration that passwords are at least 8 characters long
- Non-functional security requirement example: all user input must be sanitised before being used in database queries

Software Vulnerabilities

- Memory safety violations
    - Buffer overflows
    - Dangling pointers
- Input validation errors
    - Code injection
    - Cross-site scripting in web applications
    - Email injection
    - Format string attacks
    - HTTP header injection
- Race conditions
    - Symlink races
    - Time of check to time of use bugs
    - SQL injection
- Privilege confusion
    - Clickjacking
    - Cross-site request forgery
    - FTP bounce attack
- Side-channel attack
    - Timing attack

Common security testing approaches

- Often difficult to craft (e.g. unit tests from non-functional requirements)
- Two common approaches:
    - Test for known vulnerability types
    - Attempt directed or random search of program state space to uncover the 'weird corner cases'

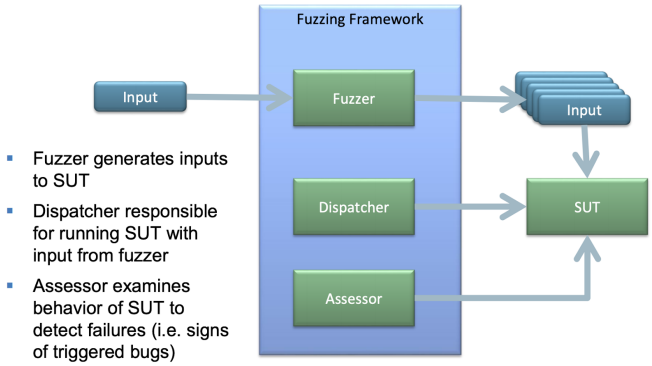
Penetration testing

- Manually try to 'break' software
- Relies on human intuition and experience
- Typically involves looking for known common problems
- Can uncover problems that are impossible or difficult to find using automated methods
    - But results completely dependent on skill of tester

Fuzz testing

- Idea: send semi-valid input to a program and observe its behaviour
    - Black-box testing - *System Under Test* (SUT) treated as a 'black-box'
    - The only feedback is the output and/or externally observable behaviour or SUT
    - First proposed in a 1990 paper where completely random data was sent to 85 common Unix utilities in 6 different systems and 24-33% crashed
        - Remember: crash implies memory protection errors
        - Crashes are often signs of exploitable flaws in the program

Fuzz testing architecture



Fuzzing components: input generation

- Simplest method: completely random
    - Won't work well in practice - input deviates too much form expected format, rejected early in processing
- Two common methods:
    - Mutation based fuzzing
    - Generation based fuzzing

Mutation based fuzzing

- Start with a valid input, and 'mutate' it
    - Flip some bits, change value of some bytes
    - Programs that have highly structured input, e.g. XML, may require 'smarter' mutations
- Challenge: how to select appropriate seed input?
    - If official test suites are available, these can be used
- Generally mostly used for programs that take files as input
    - Trickier to do when interpretation of inputs depends on program state, e.g. network protocol parsers (the way a message is handled depends on previous messages)

Pros and cons of mutation based fuzzing

Advantages

- Easy to get started
- No (or little) knowledge of specific input format needed

Disadvantages

- Typically yields low code coverage - inputs tend to deviate too much from expected format and are rejected by early sanity checks
- Hard to reach 'deeper' parts of programs by random guessing (e.g. nested control flows)

Generation based fuzzing

- Idea: use a specification of the input format (e.g. a grammar) to automatically generate semi-valid inputs
- Usually combined with various fuzzing heuristics that are known to trigger certain vulnerability types
    - Very long strings, empty strings
    - Strings with format specifiers, 'extreme' format strings
        - %n%n%n%n%n%n%n%n%n%n%n%n%n%n%n%n, %5000000.x
    - Very large or small values, values close to max or min for data type
        - 0x0, 0xffffffff, 0x7fffffff, 0x80000000, 0xfffffffe
    - Negative values where positive ones are expected

Pros and cons of generation based fuzzing

Advantages

- Input is much closer to the expected, much better cover
- Can include models of protocol state machines to send messages in the sequence expected by SUT

Disadvantages

- Requires input format to be known
- May take considerable time to write the input format grammar/specification
- Not widely applicable

Examples of generation based fuzzers (open source)

- SPIKE (2001)
    - Early successful generation based fuzzer for network protocols
    - Designed to fuzz input formats consisting of blocks with fixed or variable size. E.g. [type][length][data]
- Peach (2006)
    - Powerful fuzzing framework which allows specifying input formats in an XML format
    - Can represent complex input formats, but relatively steep learning curve
- Sulley (2008)
    - More modern fuzzing framework designed to be somewhat simpler to use than e.g. Peach.
    - Also several commercial fuzzers, e.g. Codenomicon DEFENSICS, BeStorm, etc

Fuzzing components: the dispatcher

- Responsible for running the SUT on each input generated by fuzzer module
    - Must provide suitable environment for SUT
        - E.g. implement a 'client' to communicate with a SUT using the fuzzed network protocol
    - SUT may modify the environment (file system, etc.)
        - Some fuzzing frameworks allow running SUT inside a virtual machine and restoring from known good snapshot after each SUT execution
- Takes care of how to run the system runs in the environment and setting up the environment

Fuzzing components: the assessor

- Must automatically assess observed SUT behaviour to determine if a fault was triggered
    - For C/C++ programs: monitor for memory access violations, e.g. out-of-bounds reads or writes
    - Simplest method: just check if SUT crashed
    - Problem: SUT may catch signals/exceptions to gracefully handle e.g. segmentation faults
        - Difficult to tell if a fault (which could have been exploitable with carefully crafted input) have occured

Improving fault detection

- One solution is to attach a programmable debugger to SUT
    - Can catch signals/exceptions prior to being delivered to application
    - Can also help in manual diagnosis of detected faults by recording stack traces, values of registers, etc.
- However: all faults do not result in failures, i.e. a crash or other observable behaviour (e.g. Heartbleed)
    - An out-of-bounds read/write or use-after-free may e.g. not result in a memory access violation
    - Solution: use a dynamic-analysis tool that can monitor what goes on 'under the hood'
        - Can potentially catch more bugs, but SUT runs (considerably) slower
        - Need more time for achieving the same level of coverage

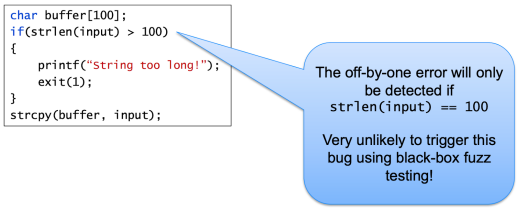
Memory error checkers

Two open source examples:

- AddressSanitizer
    - Applies instrumentation during compilation: Additional code is inserted in program to check for memory errors
    - Monitors all calls to malloc/new/free/delete – can detect if memory is freed twice, used after free, out of bounds access of heap allocated memory, etc.
    - Inserts checks that stack buffers are not accessed out of bounds ▪ Detects use of uninitialized variables
- Valgrind/Memcheck
    - Applies instrumentation directly at the binary level during runtime – does not need source code!
    - Can detect similar problems as AddressSanitizer
    - Applying instrumentation at the machine code level has some benefits – works with any build environment, can instrument third-party libraries without source code, etc.
    - But also comes at a cost; Runs slower than e.g. AddressSanitizer and can generally not detect out-of-bounds access to buffers on stack
    - Size of stack buffers not visible in machine code

Limitations of fuzz testing

- Many programs have an infinite input space and state space - combinatorial explosion!
    - You never know when to stop
- Conceptually a simple idea, but many subtle practical challenges
- Difficult to create a truly generic fuzzing framework that can cater for all possible input formats
    - For best results often necessary to write a custom fuzzer for each particular SUT
- (Semi)-randomly generated inputs are very unlikely to trigger certain faults
- Example of when fuzz testing will not work



Fuzzing outlook

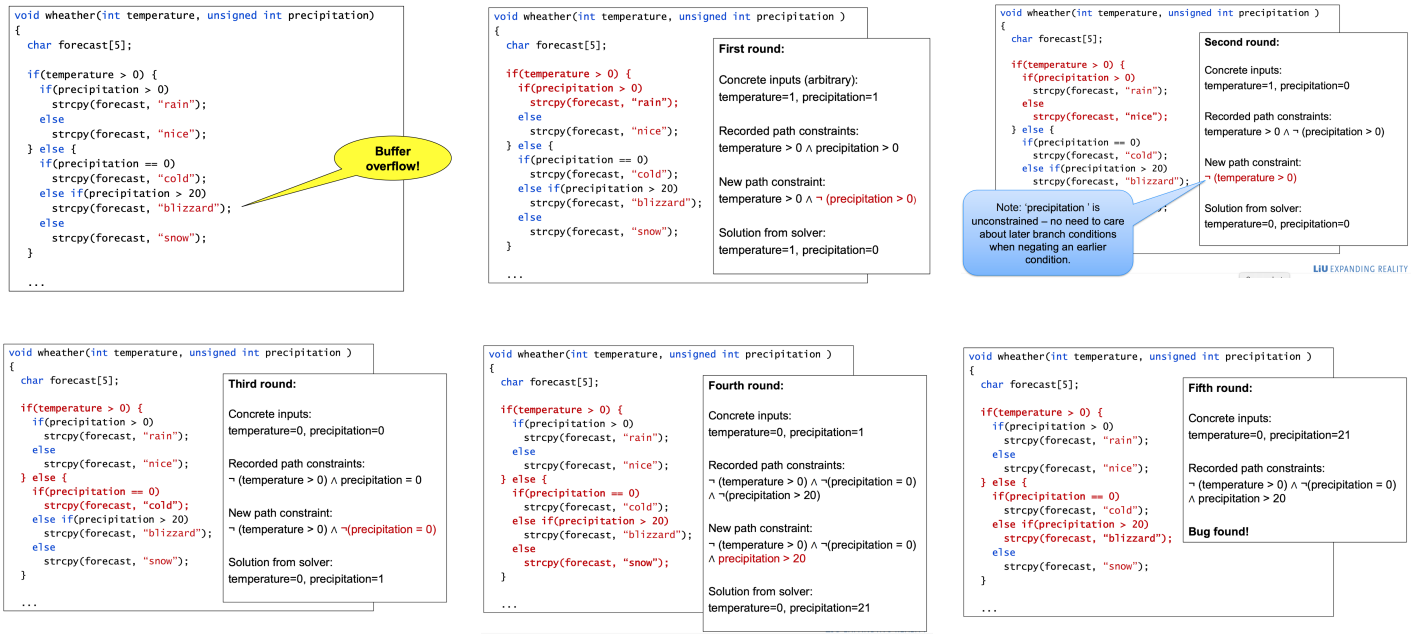
- Mutation-based fuzzing can typically only find the 'low-hanging fruit' - shallow bugs that are easy to find
- Generation-based fuzzers almost invariably gives better coverage, but requires much more manual effort

- Current research in fuzzing attempts to combine the ‘fire and forget’ nature of mutation-based fuzzing and the coverage of generation-based fuzzing
    - Evolutionary fuzzing combines mutation with genetic algorithms to try to ‘learn’ the input format automatically
        - Recent successful example is the ‘American Fuzzy Lop’ (AFL)
    - Whitebox fuzzing generates test cases based on the control-flow structure of the SUT

Concolic Testing

- Idea: combine concrete and symbolic execution
    - Concolic execution (CONCrete and symbOLIC)
- Concolic execution workflow:
    1. Execute the program for real on some input and record the path taken
    2. Encode path as query to SMT solver and negate one branch condition
    3. Ask the solver to find new satisfying input that will give a different path
- Reported bugs are always accompanied by an input that triggers the bug (generated by SMT solver)
    - Complete - reported bugs are always real bugs!

Example



Challenges with concolic testing: Path Explosion

- Number of paths increase exponentially with number of branches
    - Most real-world programs have an infinite state space
        - For example, number of loop iterations may depend on size of input
- Not possible to explore all paths
- Depth first search will easily get ‘stuck’ on one part of the program
    - May e.g. keep exploring the same loop with more and more iterations
- Breadth-first search will take a very long time to reach ‘deep’ states
    - May take ‘forever’ to reach the buggy code
- Try ‘smarter’ ways of exploring the program state space
    - May want to try to run loops many times to uncover possible buffer overflows
    - But also want to maximise coverage of different parts of the program

Generational search ('whitebox fuzzing')

- The microsoft SAGE system implements 'whitebox fuzzing'
  - Performs concolic testing, but prioritises paths based on how much they improve coverage
  - Results can be assessed similar to black-box fuzzing (with dynamic analysis tools, etc.)
- Search algorithm outline
  1. Run program on concrete seed input and record path constraint
  2. For each branch condition:
     a. Negate condition and keep the earlier conditions unchanged
     b. Have SMT solver generate new satisfying input and run program with that input
     c. Assign input a score based on how much it improves coverage (i.e. how much previously unseen code will be executed with that input)
     d. Store input in a global worklist **sorted on score**
  3. Pick input at head of worklist and repeat

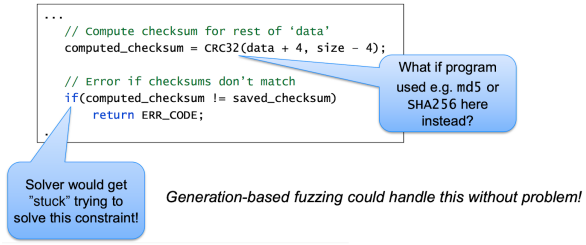
Rationale for 'whitebox fuzzing'

- Coverage based heuristic avoids getting 'stuck' as in DFS
  - If one test case executes the exact same code as in a previous test case its score will be 0, moved to end of worklist, probably never used again
  - Gives sparse but more diverse search of the paths of the program
- Implements a form of greedy search heuristic
  - For example, loops may only be executed once
  - Only generates input 'close' to the seed input (cf. mutation-based fuzzing)
    - Will try to explore the 'weird corner cases' in code exercised by seed input
    - Choice of seed input important for good coverage
- Has proven to work well in practice
  - Used in production at Microsoft to test e.g. Windows, Office, etc. prior to release
    - Has uncovered many serious vulnerabilities that was missed by other approaches (black-box fuzzing, static analysis, etc)
- Interestingly, SAGE works directly at the machine-code level
  - Note: Source code not needed for concolic execution – sufficient to collect constraints from one concrete sequence of machine-code instructions
  - Avoids hassle with different build environments, third-party libraries, programs written in different languages etc. but sacrifices some coverage due to additional approximations needed when working on machine code

Limitations of concolic testing

- The success of concolic testing is due to the massive improvement in SAT/SMT solvers during the last two decades
  - Main bottleneck is still often the solvers
  - Black-box can perform a much larger number of test cases per time unit - may be more time efficient for 'shallow' bugs (faster)
- Solving SAT/SMT problems are NP-complete
  - Solvers like Z3 use various 'tricks' to speed up common cases
  - But may take unacceptably long time to solve certain path constraints
- Solving SMT queries with non-linear arithmetic (multiplication, division, modulo, etc.) is an undecidable problem in general
  - But since programs typically use fixed-precision data types, non-linear arithmetic is decidable in this special case

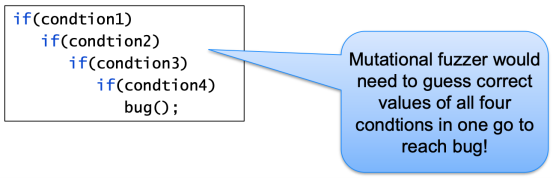
- If program uses any kind of cryptography, symbolic execution will typically fail
    - Consider previous checksum example: CRC32 is linear and reversible - solver can ‘repair’ checksum if rest of data is modified



Generation-based fuzzing could handle this without problem!

Greybox fuzzing

- Probability of hitting a ‘deep’ level of the code decreases exponentially with the ‘depth’ of the code for mutation based fuzzing
- Similarly, the time required for solving an SMT query is high, and increases exponentially with the depth of the path constraint
- Black-box fuzzing is ‘too dumb’ and whitebox fuzzing may be ‘too smart’
    - Idea of greybox fuzzing is to find a sweet spot in between



- Instead of recording full path constraint (as in whitebox fuzzing), record light-weight coverage information to guide fuzzing
- Use evolutionary algorithms to ‘learn’ input format
    - Guide the mutation towards improved coverage
- American Fuzzy Lop (AFL) is considered the current state-of-the-art in fuzzing
    - Performs ‘regular’ mutation-based fuzzing (using several different strategies) and measures code coverage
    - Every generated input that resulted in any new coverage is saved and later re-fuzzed
        - This extremely simple evolutionary algorithm allows AFL to gradually ‘learn’ how to reach deeper parts of the program
    - Also highly optimised to speed - can reach several thousand test cases per second
        - This often beats smarter (and slower) methods like whitebox fuzzing!

Conclusions

- Fuzzing - black-box testing method
    - Semi-random input generation
    - Despite being a brute-force, somewhat ad-hoc approach to security testing, experience has shown that it improves security in practice
- Concolic testing - whte-box testing method
    - Input generated from control-structure of code to systematically explore different paths of the program
    - Some in-house and academic implementations exist, but some challenges left to solve before widespread adoption
- Greybox fuzzing
    - Coverage-guided semi-random input generation

    - High speed sometimes beats e.g. concolic testing, but shares some limitations with mutation-based fuzzing (e.g. magic constants, checksums)
- Test cases generated automatically, either semi-randomly or from structure of code
    - Test cases not based on requirements
    - Pro: not ‘biased’ by developers’ view of ‘how things should work’, can uncover unsound assumptions or corner cases not covered by specification
    - Con: fuzzing and concolic testing mostly suited for finding *implementation* errors, e.g. buffer overflows, arithmetic overflows, etc.
- Generally hard to test for high-level errors in requirements and design using these methods
- Different methods are good at finding different kinds of bugs, but none is a silver bullet
    - Fuzzing cannot be the only security assurance method used
    - Static analysis
    - Manual reviews (code, design documents, etc.)
    - Regular unit/integration/system testing

Lecture 14 - Integration and Component-based testing

What is integration testing?

	Module test	Integration test	System test
Specification	Module interface	Interface specifications, module breakdown	Requirements specification
Visible structure	Coding details	Modular structure (software architecture)	None
Scaffolding required	Some	Often extensive	Some
Looking for faults in	Modules	Interactions, compatibility	System functionality

Integration vs Unit Testing

- Unit (module) testing is a necessary foundation
  - Unit level has maximum controllability and visibility
  - Integration testing can never compensate for inadequate unit testing
- Integration testing may serve as a process check
  - If module faults are revealed in integration testing, they signal inadequate unit testing
  - If integration faults occur in interfaces between correctly implemented modules, the errors can be traced to module breakdown and interface specifications

Integration Faults

- Inconsistent interpretation of parameters or values
  - Example: mixed units (meters/yards) in Martian Lander
- Violations of value domains, capacity, or size limits
  - Example: buffer overflow
- Side effects on parameters or resources
  - Example: conflict on (unspecified) temporary file
- Omitted or misunderstood functionality
  - Example: inconsistent interpretation of web hits
- Non Functional properties
  - Example: unanticipated performance issues
- Dynamic mismatches
  - Example: incompatible polymorphic method calls

Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
static void ssl io filter disable(ap filter t *f)
{   bio filter in ctx t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter ctx->pssl
}
```

The missing code is for a **structure defined and created elsewhere**, accessed through an opaque pointer.

Almost impossible to find with unit testing (inspection and some dynamic techniques could have found it)

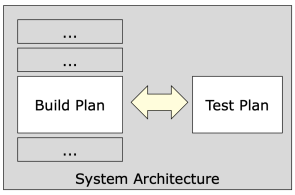
Common misconception

“Yes, I implemented module A but I didn’t test it thoroughly yet. It will be tested along with module B when that’s ready.”

Translation: “I didn’t think at all about the strategy for testing. I didn’t design module A for testability and I didn’t think about the best order to build and test modules A and B.”

Integration Plan + Test Plan

- Integration test plan drives and is driven by the project ‘build plan’
    - A key feature of the system architecture and project plan

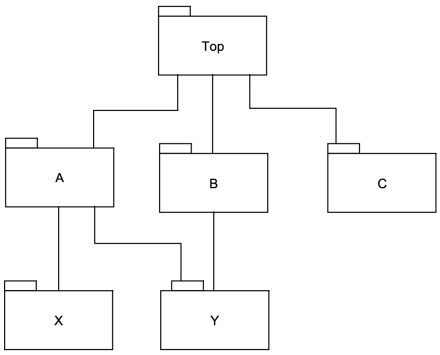


Big Bang Integration Test

- An extreme and desperate approach: test only after integrating all modules
- Pro: does not require scaffolding (the only excuse, and a bad one!)
- Cons: minimum observability, diagnosability, efficacy, feedback and high cost of repair (cost of repairing a fault rises as function of time between error and repair)

Structural and functional strategies

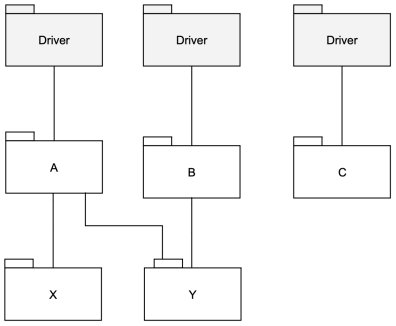
- **Structural orientation:** modules constructed, integrated and tested based on a hierarchical project structure
    - Top-down



Working from the top level (in terms of ‘use’ or ‘include’ relation) toward the bottom. No drivers required if the program is tested from top-level interface (e.g. GUI, CLI, web app)

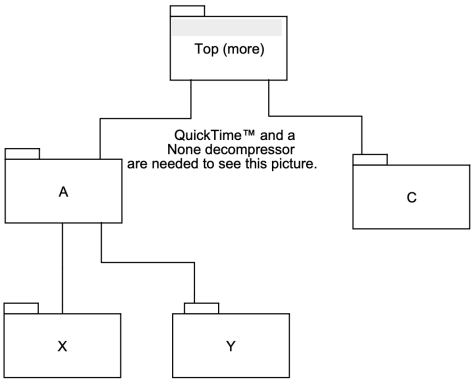
- Write stubs of called or used modules at each step in construction (after stubbing A, you realise you need stubs for X and Y)
- As modules replace stubs, more functionality is testable
- Until the program is complete, and all functionality can be tested

    - Bottom-up



- Starting at the leaves of the ‘uses’ hierarchy, we never need stubs (X, Y)
- But we must construct drivers for each module (as in unit testing)
- An intermediate module replaces a driver, and it needs its own driver (A)
- So we may have several working subsystems
- Driver: dummy functionality that calls a module

- Sandwich

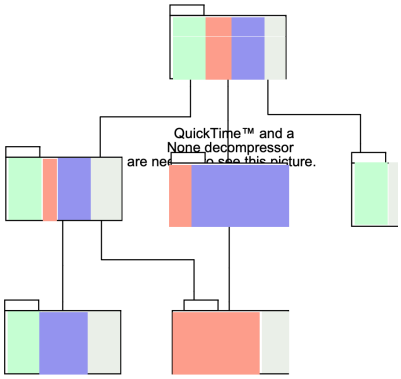


Working from the extremes (top and bottom) toward the center, we may use fewer drivers and stubs.

- It is a combination of top-down and bottom-up
- Sandwich integration is flexible and adaptable, but complex to plan

- Backbone

- **Functional orientation:** modules integrated according to application characteristics or features
  - Thread



A ‘thread’ is a portion of several modules that together provide a user-visible program feature.

- By integrating one thread, then another, we maximise visibility for the user
- As in sandwich integration testing, we can minimise stubs and drivers, but the integration plan may be complex

  - Critical modules
- Strategy: start with riskiest modules
    - Risk assessment is necessary first step
    - May include technical risks (is X feasible?)process risks (is the schedule for X realistic?), other risks
  - May resemble thread or sandwich process in tactics for flexible build order
    - E.g. constructing parts of one module to test functionality in another
  - Key point is risk-oriented process
    - Integration testing as a risk-reduction activity, designed to deliver any bad news as early as possible

Choosing a strategy

- Functional strategies require more planning
  - Structural strategies (bottom-up, top-down, sandwich) are simpler
  - Might not work well with larges systems
  - But thread and critical modules testing provide better process visibility, especially in complex systems
- Possible to combine
  - top-down, bottom-up or sandwich are reasonable for relatively small components and subsystems
  - Combinations of thread and critical modules integration testing are often preferred for larger subsystems

Working definition of *Component*

- **Reusable** unit of **deployment** and **composition**
    - Deployed and integrated multiple times
    - Integrated by different teams (usually)
        - Component producer is distinct from component user
- Characterised by an *interface* or *contract*
    - Describes access points, parameters, and all functional and non-functional behaviour and conditions for using the component
    - No other access (e.g. source code) is usually available
- Often larger grain than objects or packages
    - Example: a complete database system may be a component

Components - Related Concepts

- Framework
    - Skeleton or micro-architecture of an application
    - May be packaged and reused as a component, with ‘hooks’ or ‘slots’ in the interface contract
- Design patterns
    - Logical design fragments
    - Frameworks often implement patterns, but patterns are not frameworks
    - Frameworks are concrete, patterns are abstract
- Component-based system
    - A system composed primarily by assembling components, often ‘commercial off-the-shelf’ (COTS) components
    - Usually includes application-specific ‘glue code’

Component Interface Contracts

- Application programming interface (API) is distinct from implementation
    - Example: DOM interface for XMLis distinct from many possible implementations, from different sources
- Interface includes *everything that* must be known to use the component
    - More than just method signatures, exceptions, etc.
    - May include non-functional characteristics like performance, capacity, security
    - May include dependence on other components

Challenges in Testing Components

- The component builder’s challenge:
    - Impossible to know all the ways a component may be used
    - Difficult to recognise and specify all potentially important properties and dependencies
- The component user’s challenge:
    - No visibility ‘inside’ the component
    - Often difficult to judge suitability for a particular use and context

Testing a component: producer view

- First: thorough unit and subsystem testing
    - Includes thorough functional testing based on application program interface (API)
    - Rule of thumb: reusable component requires at least twice the effort in design, implementation, and testing as a subsystem constructed for a single use (often more)

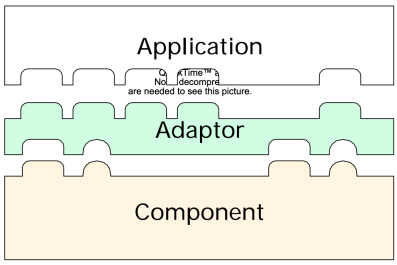
- Second: thorough acceptance testing
    - Based on scenarios of expected use
    - Includes stress and capacity testing
        - Find and document the limits of applicability

Testing a component: user view

- Not primarily to find faults in the component
- Major question: is the component suitable for *this* application?
    - Primary risk is not fitting the application context
        - Unanticipated dependence or interactions with environment
        - Performance or capacity limits
        - Missing functionality, misunderstood APi
    - Risk high when using component for first time
- Reducing risk: trial integration early
    - Often worthwhile to build driver to test model scenarios, long before actual integration

Adapting and Testing a Component

- Applications often access components through an adaptor, which can also be used by a test driver



Summary

- Integration testing focuses on interactions
    - Must be built on foundation of thorough unit testing
    - Integration faults often traceable to incomplete or misunderstood interface specifications
- Prefer prevention to detection and make detection easier Prefer prevention to detection, and make detection easier by imposing design constraints
- Strategies tied to project build order
    - Order construction, integration, and testing to reduce cost or risk
- Reusable components require special care
    - For component builder, and for component user

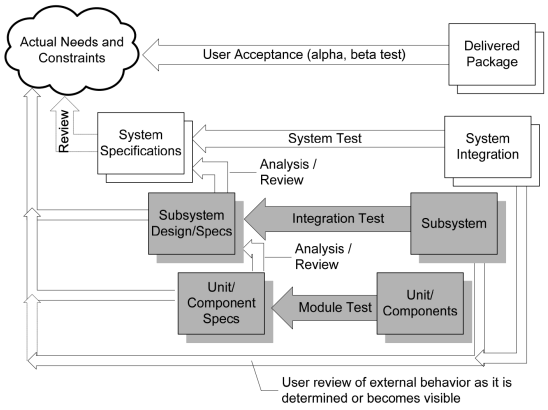
Lecture 14.5 - Testing Object Oriented Software

Characteristics of OO Software

Typical OO software characteristics that impact testing

- State dependent behaviour
- Encapsulation
- Inheritance
- Polymorphism and dynamic binding
- Abstract and generic classes
- Exception handling

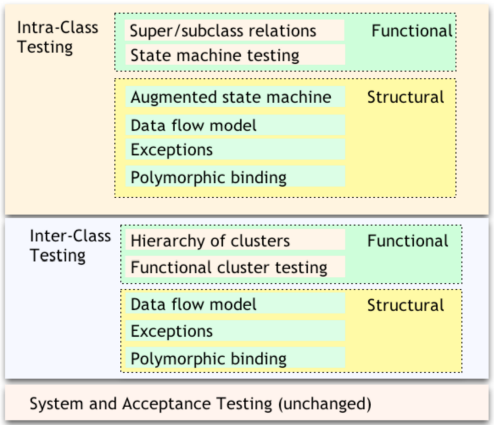
Quality activities and OO Software



OO definitions of unit and integration testing

- Procedural software
  - Unit is a single program, function or procedure
  - More often: a unit of work that may correspond to one or more intertwined functions or programs
- Object oriented software
  - Unit is a class or (small) cluster of strongly related classes (e.g. sets of Java classes that correspond to exceptions)
  - Unit testing is intra-class testing
  - Integration testing is inter-class testing (cluster of classes)
  - Dealing with single methods separately is usually too expensive (complex scaffolding), so methods are usually tested in the context of the class they belong to

Orthogonal approach: stages



Intraclass state machine testing

- Basic idea:
  - The state of an object is modified by operations
  - Methods can be modeled as state transitions
  - Test cases are sequences of method calls that traverse the state machine model
- State machine model can be derived from specification (functional testing), code (structural testing), or both

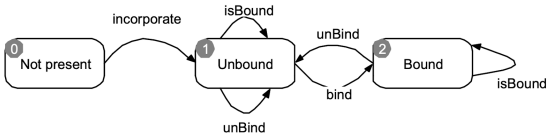
Informal state-full specifications

- Slot: represents a slot of a computer model. Slots can be bound or unbound.
- Bound slots are assigned a compatible component, unbound slots are empty
- Class slot offers the following services:
  - Install: slots can be installed on a model as required or optional.
  - Bind: slots can be bound to a compatible component
  - Unbind: bound slots can be unbound by removing the bound component
  - isBound: returns the current binding, if bound; otherwise returns the special value empty

Identifying states and transitions

- From the informal specification we can identify three states:
  - Not installed
  - Unbound
  - Bound
- And four transitions
  - Install: from not installed to unbound
  - Bind: from unbound to bound
  - Unbind: to unbound
  - isBound: does not change state

Deriving an FSM and test cases



- TC-1: incorporate, isBound, bind, isBound
- TC-2: incorporate, unBind, bind, unBind, isBound

Testing with state diagrams

- A statechart (called a ‘state diagram’ in UML) may be produced as part of a specification or design
  - May also be implied by a set of message sequence charts (interaction diagrams) or by modeling formalisms
- Two options:
  - Convert (‘flatten’) into standard finite-state machine, then derive test cases
  - Use state diagram model directly

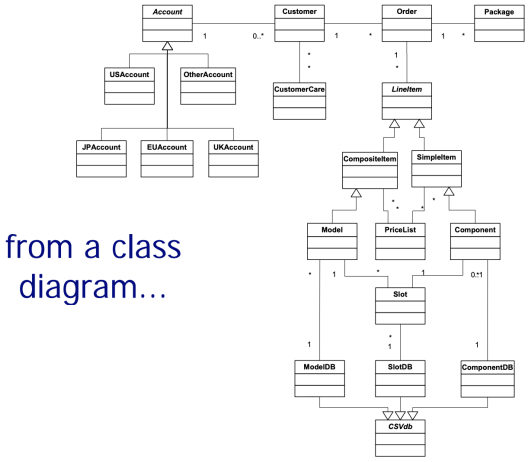
Interclass testing

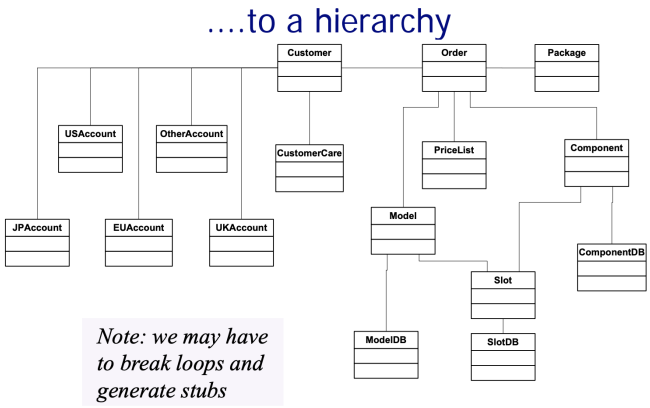
- The first level of *integration testing* for object oriented software
  - Focus on interactions between classes

- Bottom-up integration according to ‘depends’ relation
    - A depends on B: build and test B, then A
- Start from use/include hierarchy
    - Implementation-level parallel to logical ‘depends’ relation
        - Class A makes method calls on class B
        - Class A objects include references to class B methods
            - But only if reference means ‘is part of’

The use-include hierarchy

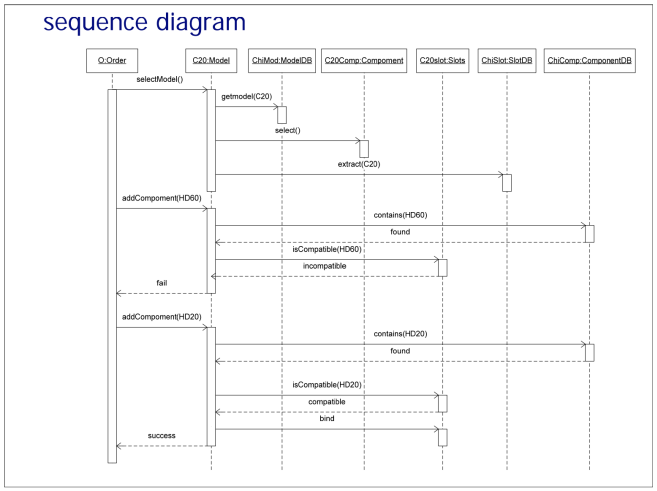
- Shows interactions between the classes





Interactions in Interclass Tests

- Proceed bottom-up
- Consider all combinations of interactions
    - Example: a test case for class *Order* includes a call to a method of class *Model*, and the called method calls a method of class *Slot*, exercise all possible relevant states of the different classes
    - Problem: combinatorial explosion of cases
    - So select a subset of interactions
        - Arbitrary or random selection
        - Plus all significant interaction scenarios that have been previously identified in design and analysis: sequence and collaboration diagrams



Using structural information

- Start with functional testing
  - As for procedural software, the specification (formal or informal) is the first source of information for testing object-oriented software
    - “Specification” widely construed: Anything from a requirements document to a design model or detailed interface description
- Then add information from the code (structural testing)
  - Design and implementation details not available from other sources

From the implementation…

```
public class Model  extends Orders.CompositeItem {
....
   private boolean legalConfig = false; // memoized
....
   public boolean isLegalConfiguration() {
    if (! legalConfig) {
       checkConfiguration();
    }
    return legalConfig;
   }
.....
   private void checkConfiguration() {
    legalConfig = true;
    for (int i=0; i < slots.length; ++i) {
       Slot slot = slots[i];
       if (slot.required && ! slot.isBound()) {
          legalConfig = false;
       } ...}  ...  }
```

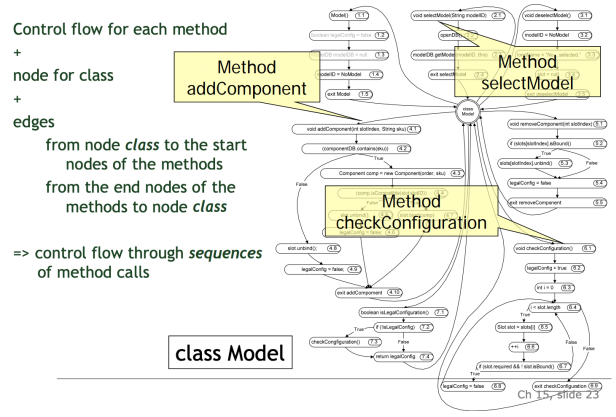
private instance variable

private method

Intraclass data flow testing

- Exercise sequences of methods
  - From setting or modifying a field value
  - To using that field value
- We need a control flow graph that encompasses more than a single method

The intraclass control flow graph



Interclass structural testing

- Working bottom up in dependence hierarchy “bottom up” in dependence hierarchy
  - Dependence is not the same as class hierarchy; not always the same as call or inclusion relation
  - May match bottom-up build order
  - Starting from leaf classes, then classes that use leaf classes, ...
  - Summarize effect of each method: Changing or using, object state, or both
    - Treating a whole object as a variable (not just primitive types)

Lecture 15 - Polymorphism and Dynamic Binding

Isolated calls: the combinatorial explosion problem

How do you test an abstract class with multiple bindings?

```
abstract class Credit {
...
  abstract boolean validateCredit( Account a, int amt, CreditCard c);
...
}
```

EduCredit	USAccount	VISACard
BizCredit	UKAccount	AmExpCard
IndividualCredit	EUAccount	StoreCard
	JPAccount	
	OtherAccount	

The combinatorial problem: 3 x 5 x 3 = 45 possible combinations of dynamic bindings (just for this one method!)

The combinatorial approach

Take each parameter and only test combinations with one other parameter.

Identify a set of combinations that cover all pairwise combinations of dynamic bindings

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard

Same motivation as pairwise specification-based testing

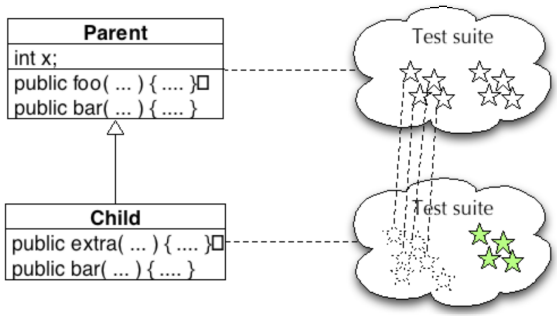
Inheritance

- When testing a subclass
  - We would like to re-test only what has not been thoroughly tested in the parent class
    - For example, no need to test hashCode and getClass methods inherited from class Object in Java
  - But we should test any method whose behavior may have changed
    - Even accidentally!

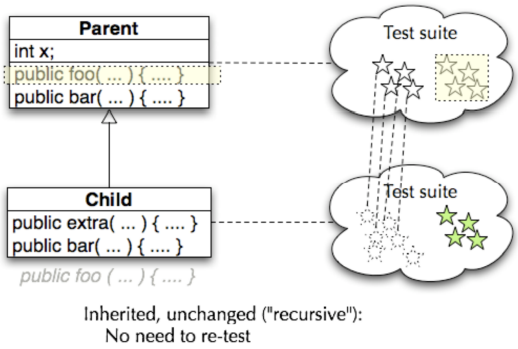
Reusing Tests with the Testing History Approach

- Track test suites and test executions
  - Determine which new tests are needed
  - Determine which old tests must be re-executed
- New and changed behaviour
  - New methods must be tested
  - Redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
  - Other inherited methods do not have to be retested

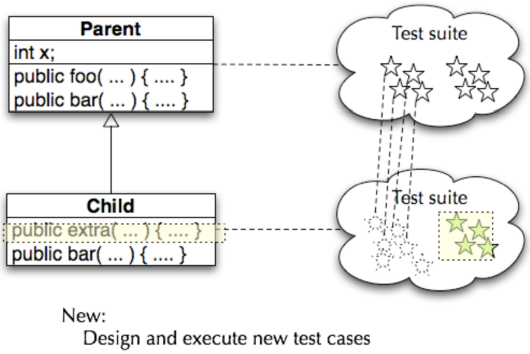
Testing history



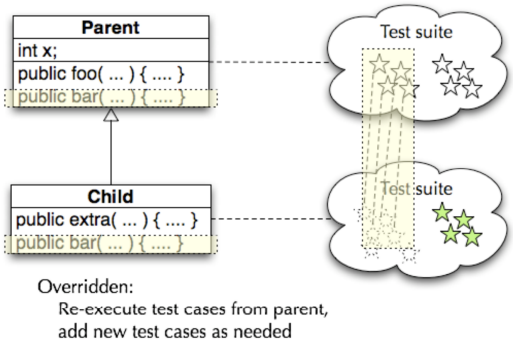
Test history



Inherited, unchanged (`foo()`)



Newly introduced methods (`extra()`)

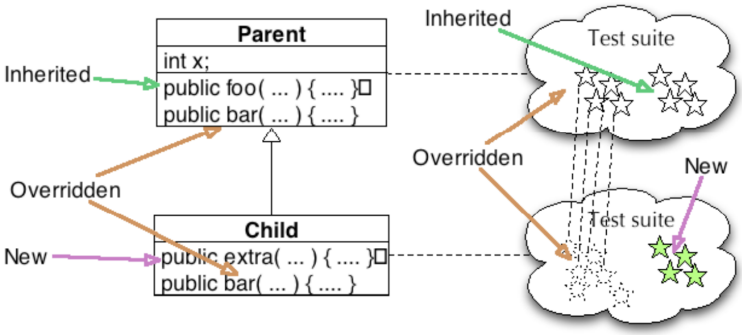


Overridden methods (`bar()`)

Testing history - some details

- Abstract methods (and classes)
  - Design test cases when abstract method was introduced (even if it can't be executed yet)
- Behaviour changes
  - Should we consider a method 'redefined' if another new or redefined method changes its behaviour?
    - The standard 'testing history' approach does not do this
    - It might be reasonable combination of data flow (structural) OO testing with the (functional) testing history approach

Testing history overview diagram



Does testing history help?

- Executing test cases should (usually) be cheap
  - It may be simpler to re-0execute the full test suite of the parent class
  - But still add to it for the same reasons
- But sometimes execution is not cheap
  - Examples
    - Control of physical devices
    - Very large test suites
      - E.g. some Microsoft product test suites require more than one night (so daily build cannot be fully tested)
    - Then some use of testing history is profitable

Exception handling

```
void addCustomer(Customer theCust) {
   customers.add(theCust);
    }
    public static Account
   newAccount(...)
   throws InvalidRegionException
    {
   Account thisAccount = null;
   String regionAbbrev = Regions.regionOfCountry(
                    mailAddress.getCountry());
   if (regionAbbrev == Regions.US) {
       thisAccount = new USAccount();
   } else if (regionAbbrev == Regions.UK) {
       ....
   } else if (regionAbbrev == Regions.Invalid) {
       throw new
   InvalidRegionException(mailAddress.getCountry());
```

exceptions create implicit control flows and may be handled by different handlers

Test exception handling

- Impractical to treat exceptions like normal flow
  - Too many flows: every array subscript reference, every memory allocation, every case
  - Multiplied by matching them to every handler that could appear immediately above them on the call stack
  - Many actually impossible
- So we separate testing exceptions
  - Ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: each exception handler, and each explicit throw or rethrow of an exception

Summary

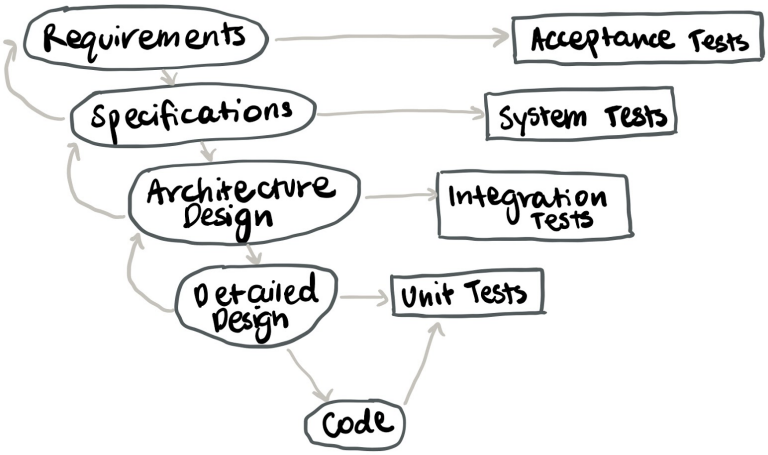
- Several features of object-oriented languages and programs impact testing
  - From encapsulation and state-dependent structure to generics and exceptions
  - But only at unit and subsystem levels
  - And fundamental principles are still applicable
- Basic approach is orthogonal
  - Techniques for each major issue (e.g., exception handling generics inheritance) can be applied handling, generics, inheritance, ...) can be applied incrementally and independently

Lecture 16 - System, Acceptance, and Regression Testing

What tests what?

	System	Acceptance	Regression
Test for	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by	Development test group	Test group with users	Development test group
	Verification	Validation	Verification

V Development Process



System testing

- Key characteristics:
  - Comprehensive (the whole system, the whole specification)
  - Based on specification of observable behavior
    - Verification against a requirements specification, not validation, and not opinions
  - Independent of design and implementation
    - Independence: avoid repeating software design errors in system design

Independent V&V

- One strategy for maximising independence: system (and acceptance) test performed by a different organisation
  - Organisationally isolated from developers (no pressure to say ‘ok’)
  - Sometimes outsourced to another company of agency
    - Especially for critical systems
    - Outsourcing for independent judgement, not to save money
    - May be additional system test, not replacing internal V&V
  - Not all outsourced testing is IV&V
    - Not independent if controlled by development organisation

Independence without changing staff

- If the development organisation controls system testing
  - Perfect independence may be unattainable, but we can reduce undue influence

- Develop system test cases early
    - As part of requirements specification, before major design decisions have been made
        - Agile ‘test first’ and conventional ‘V model’ are both examples of designing system test cases before designing the implementation
        - An opportunity for ‘design for test’: structure system for critical system testing early in project
- Different group in your company will write the system tests and the group should not know how the implementation was done

Incremental System Testing

- System tests are often used to measure progress
    - System test suite covers all features and scenarios of use
    - As project progresses, the system passes more and more system tests
- Assumes a ‘threaded’ incremental build plan: features exposed at top level as they are developed

Global Properties

- Some system properties are inherently global
    - Performance, latency, reliability
    - Early and incremental testing is still necessary, but provide only estimates
- A major focus of system testing
    - The only opportunity to verify global properties against actual system specifications
    - Especially to find unanticipated effects, e.g. an unexpected performance bottleneck

Context-Dependent Properties

- Since you depend on system context and how it is used, you will need to study the environment, what type of users use it, what type of network is used, what kind of bandwidth you have, how many users will use it, what is the expected response rate
- Beyond system-global: some properties depend on the system context and use
    - Examples
        - Performance properties depend on environment and configuration
        - Privacy depends both on system and how it is used
            - Medical records system must protect against unauthorised use, and authorisation must be provided only as needed
        - Security depends on threat profiles
            - And threats change!
- Testing is just one part of the approach

Establishing an Operational Envelope

- Envelope: limitations of use of the program
- Make sure to cover all usage scenarios
- When a property (e.g. performance or real-time response) is parameterised by use (requests per second, size of database), extensive stress testing is required
    - Varying parameters within the envelope, near the bounds, and go beyond
- Goal: a well-understood model of how the property varies with the parameter
    - How sensitive is the property to the parameter?
    - Where is the ‘edge of the envelope’?
    - What can we expect when the envelope is exceeded?

Stress Testing

- Often requires extensive simulation of the execution environment
    - With systematic variation: what happens when we push the parameters? What if the number of users or requests is 10 times more, or 1000 times more?
- Often requires more resources (human and machine) than typical test cases
    - Separate from regular feature tests
    - Run less often, with more manual control
    - Diagnose deviations from expectation
        - Which may include difficult debugging of latent faults

Capacity Testing

- How much can the system cope with?
- When: systems that are intended to cope with high volumes of data should have their limits tested and we should consider how they fail when capacity is exceeded
- What/How: usually we will construct a harness that is capable of generating a very large volume of simulated data that will test the capacity of the system or use existing records
- Why: we are concerned to ensure that the system is fit for purpose say ensuring that a medical records system can cope with records for all people in the UK (for example)
- Strengths: provides some confidence the system is capable of handling high capacity
- Weaknesses: simulated data can be unrepresentative; can be difficult to create representative tests; can take a long time to run

Security Testing

- How vulnerable your system is to malicious attacks and whether you have protection against them
- When: most systems that are open to the outside world and have a function that should not be disrupted require some kind of security test. Usually we are concerned to thwart malicious users.
- What/How: there are a range of approaches. One is to use league tables of bugs/errors to check and review the code (e.g. SANS top twenty-five security related programming errors). We might also form a team that attempts to break/break into the system.
- Why: some systems are essential and need to keep running, e.g. the telephone system, some systems need to be secure to maintain reputation.
- Strengths: this is the best approach we have most of the effort should go into design and the use of known secure components.
- Weaknesses: we only cover known ways in using checklists and we do not take account of novelty using a team to try to break does introduce this.

Performance Testing

- Provide a service-level agreement on system performance that should always be satisfied
- When: many systems are required to meet performance targets laid down in a service level agreement (e.g. does your ISP give you 2Mb/s download?)
- What/How: there are two approaches - modelling/simulation, and direct test in a simulated environment (or in the real environment)
- Why: often a company charges for a particular level of service - this may be disputed if the company fails to deliver. E.g. The VISA payments system guarantees 5s authorisation time delivers faster and has low variance. Customers would be unhappy with less
- Strengths: can provide good evidence of the performance of the system, modelling can identify bottlenecks and problems
- Weaknesses: issues with how representative tests are

Compliance Testing

- Every system you release should work against certain architectures (e.g. software on multiple OSes)
- When: we are selling into a regulated market and to sell we need to show compliance. E.g. if we have a C compiler we should be able to show it correctly compiles ANSI C
- What/How: often there will be standardised test sets that constitute good coverage of the behaviour of the system (e.g. a set of C programs, and the results of running them)
- Why: we can identify the problem areas and create tests to check that set of conditions
- Strengths: regulation shares the cost of tests across many organisations so we can develop a very capable test set
- Weaknesses: there is a tendency for software producers to orient towards the compliance test set and do much worse on things outside the compliance test set

Documentation Testing

- Make sure that the documentation corresponds to the system you built and there are no bugs associated to it
- When: most systems that have documentation should have it tested and should be tested against the real system. Some systems embed test cases in the documentation and using the doc tests is an essential part of a new release
- What/How: test set is maintained that verifies the doc set matches the system behaviour. Could also just get someone to do the tutorial and point out the errors
- Why: the user gets really confused if the system does not conform to the documentation
- Strengths: ensures consistency
- Weaknesses: not particularly good on checking consistency of narrative rather than examples

Acceptance Testing

- Checking against customer requirements
- Guide to whether or not the current system is acceptable (in terms of quality) to be used by real users
- Has quantitative goals: availability, dependability, reliability, mean time to failure, etc.

Estimating Dependability

- Measuring quality, not searching for faults
    - Fundamentally different goal than systematic testing
- Quantitative dependability goals are statistical: reliability, availability, mean time to failure
- Requires valid statistical samples from operational profile
    - Fundamentally different from systematic testing

Definitions

- Reliability: survival probability; probability to operate without failure in a critical category
    - When function is critical during the mission time
- Availability: the fraction of time a system meets its specification
    - Good when continuous service is important but it can be delayed or denied
- Failsafe: system fails to a known safe state
    - Does it fail gracefully?
- Dependability: generalisation - system does the right thing at the right time
    - What is the probability of trust in an operation being carried out by the system?

Statistical sampling

- We need a valid operational profile (model)
  - Sometimes from an older version of the system
  - Sometimes from operational environment (e.g. for an embedded controller)
  - Sensitivity testing reveals which parameters are most important, and which can be rough guesses
- And a clear, precise definition of what is being measured
  - Failure rate: per session, per hour, or per operation?
- And many, many random samples
  - Especially for high reliability measures

System Reliability

- The reliability, $R_F(t)$ of a system is the probability that no fault of the class F occurs (i.e. system survives) during time t.

  $$R_F(t) = P(t_{init} \leq t < t_f \forall f \in F)$$

  where t_{init} is time of introduction of the system to service,
  t_f is time of occurrence of the first failure f drawn from F.

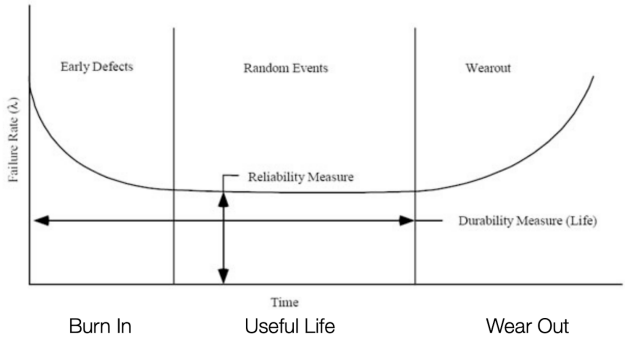
- Failure Probability, $Q_F(t)$ is complementary to $R_F(t)$

  $$R_F(t) + Q_F(t) = 1$$

- We can take off the F subscript from $R_F(t)$ and $Q_F(t)$

- When the lifetime of a system is exponentially distributed, the reliability of the system is: $R(t) = e^{-\lambda t}$ where the parameter λ is called the failure rate

Component Reliability Model (Bucket curve for failure rate)



During useful life, components exhibit a constant failure rate λ. Reliability of a device can be modelled using an exponential distribution $R(t) = e^{-\lambda t}$

Component failure rate

- Failure rates often expressed in failures/million operating hours

Automotive Embedded System Component	Failure Rate λ
Military Microprocessor	0.022
Typical Automotive Microprocessor	0.12
Electric Motor Lead/Acid battery	16.9
Oil Pump	37.3
Automotive Wiring Harness (luxury)	775

Mean Time To Failure (MTTF)

What is the time between the start of the operation to the first failure?

- **MTTF**: Mean Time to Failure or Expected Life
- **MTTF**: Mean Time To (first) Failure is defined as the expected value of t

$$MTTF = E(t_f) = \int_0^\infty R(t)dt = \frac{1}{\lambda}$$

where λ is the failure rate.

- **MTTF** of a system is the expected time of the first failure in a sample of identical initially perfect systems.
- **MTTR**: Mean Time To Repair is defined as the expected time for repair.
- **MTBF**: Mean Time Between Failure

Serial System Reliability

The failure rate of a serial system is always worse than the failure rate of its individual components.
If one of the components fails, the entire system fails!

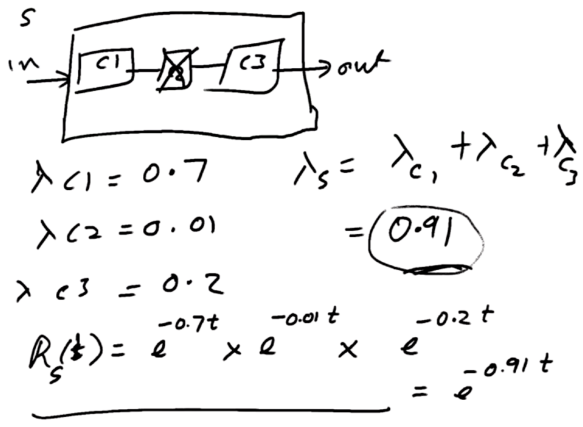
- Serially Connected Components
- $R_k(t)$ is the reliability of a single component k: $R_k(t) = e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.
- The overall system reliability $R_{ser}(t)$

$$R_{ser}(t) = R_1(t) \times R_2(t) \times R_3(t) \times \ldots \times R_n(t)$$

$$R_{ser}(t) = \prod_{i=1}^{n} R_i(t)$$

- No redundancy: Overall system reliability depends on the proper working of each component $R_{ser}(t) = e^{-t(\sum_{i=1}^{n} \lambda_i)}$
- Serial failure rate $\lambda_{ser} = \sum_{i=1}^{n} \lambda_i$

Example from lecture



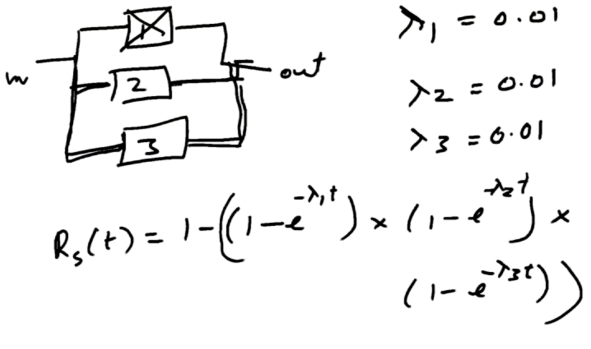
Parallel System Reliability

- Parallel Connected Components
- $Q_k(t)$ is $1 - R_k(t)$: $Q_k(t) = 1 - e^{-\lambda_k t}$
- Assuming the failure rates of components are statistically independent.

$$Q_{par}(t) = \prod_{i=1}^{n} Q_i(t)$$

- Overall system reliability: $R_{par}(t) = 1 - \prod_{i=1}^{n}(1 - R_i(t))$

Example from lecture

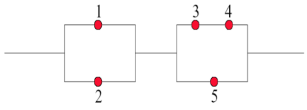


System Reliability

- Building a reliable serial system is extraordinarily difficult and expensive.
- For example: if one is to build a serial system with 100 components each of which had a reliability of 0.999, the overall system reliability would be

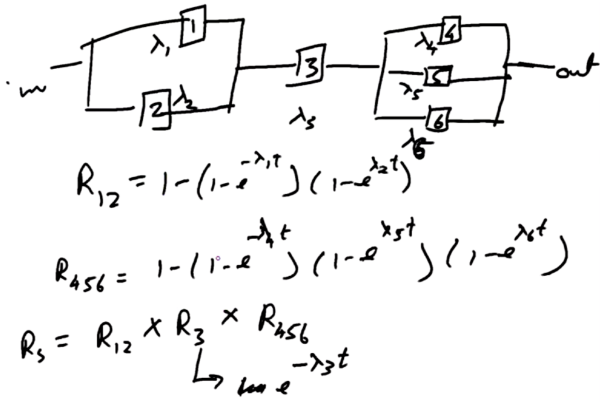
$$0.999^{100} = 0.905$$

- Reliability of System of Components



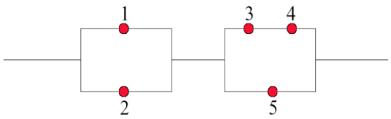
- Minimal Path Set:
  Minimal set of components whose functioning ensures the functioning of the system: {1,3,4} {2,3,4} {1,5} {2,5}

Example from lecture



Example from slides

- Parallel and Serial Connected Components



- Total reliability is the reliability of the first half, in serial with the second half.
- Given R1=0.9, R2=0.9, R3=0.99, R4=0.99, R5=0.87
- $R_t = (1-(1-0.9)(1-0.9))(1-(1-0.87)(1-(0.99 \times 0.99))) = 0.987$

Is statistical testing worthwhile?

- Necessary for critical systems (safety critical, infrastructure)
- But difficult or impossible when:
  - Operational profile is unavailable or just a guess
    - Often for new functionality involving human interaction
      - But we may factor critical functions from overall use to obtain a good model of only the critical properties
    - Reliability requirement is very high
      - Required sample size (number of test cases) might require years of test execution
      - Ultra-reliability can seldom be demonstrated by testing

Process-based measures

- Less rigorous than statistical testing
  - Based on similarity with prior projects
- System testing process - expected history of bugs found and resolved
- Alpha-beta testing
  - Alpha testing: real users, controlled environment
  - Beta testing: real users, real (uncontrolled) environment
  - May statistically sample users rather than uses
  - Expected history of bug reports

Usability Testing

- You want to test the user interface of your system (is it easy enough to understand?) through a simulator
- When: where the system has a significant user interface and it is important to avoid user error — e.g. this could be a critical application e.g. cockpit design in an aircraft or a consumer product that we want to be an enjoyable system to use or we might be considering efficiency (e.g. call-centre software)
- What/How: we could construct a simulator in the case of embedded systems or we could just have many users try the system in a controlled environment. We need to structure the test with clear objectives (e.g. to reduce decision time,...) and have good means of collecting and analysing data
- Why: there may be safety issues, we may want to produce something more useful than competitors' products, etc.
- Strengths: in well-defined contexts this can provide very good feedback – often underpinned by some theory e.g. estimates of cognitive load
- Weaknesses: some usability requirements are hard to express and to test, it is possible to test extensively and then not know what to do with the data

Reliability Testing

- You want to guarantee that the system will only fail so many times in a certain number of hours of operation
- When: we may want to guarantee some system will only fail very infrequently (e.g. nuclear power control software we might claim no more than one failure in 10,000 hours of operation). This is particularly important in telecommunications
- What/How: we need to create a representative test set and gather enough information to support a statistical claim (system structured modelling supports demonstrating how overall failure rate relates to component failure rate)
- Why: we often need to make guarantees about reliability in order to satisfy a regulator or we might know that the market leader has a certain reliability that the market expects
- Strengths: if the test data is representative this can make accurate predictions
- Weaknesses: we need a lot of data for high-reliability systems, it is easy to be optimistic

Availability/Reparability Testing

- How fast can your system go back up once a failure occurs (long down times are unideal)
- When: we are interested in avoiding long down times we are interested in how often failure occurs and how long it takes to get going again. Usually this is in the context of a service supplier and this is a Key Performance Indicator
- What/How: similar to reliability testing – but here we might seed errors or cause component failures and see how long they take to fix or how soon the system can return once a component is repaired. • Why: in providing a critical service we may not want long interruptions (e.g. 999 service). • Strengths: similar to reliability
- Weaknesses: similar to reliability – in the field it may be much faster to fix common problems because of learning

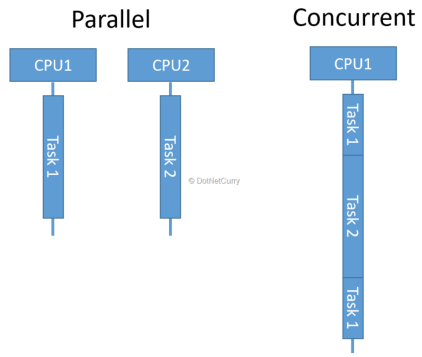
Summary

- There are a very wide range of potential tests that should be applied to a system
- Not all systems require all tests
- Managing the test sets and when they should be applied is a very complex task
- The quality of test sets is critical to the quality of a running implementation

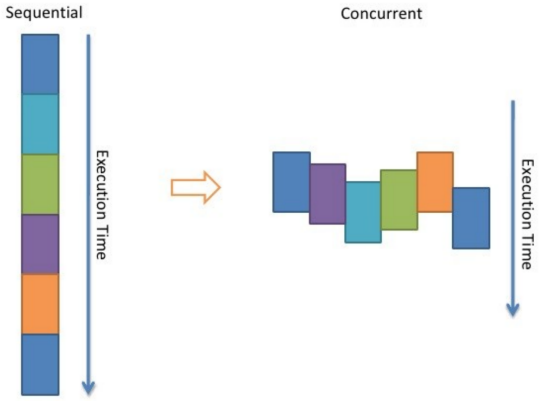
Lecture 16 - Concurrency Bugs

Concurrent programs

- Multi-core computers are common
- More programmers are having to write concurrent programs
- Concurrent programs have different bugs than sequential programs



Multi-threaded programming



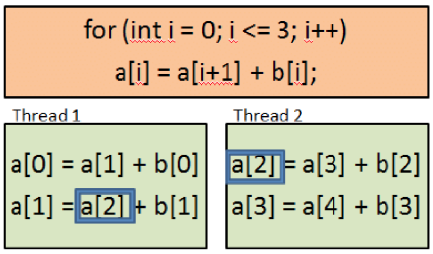
Concurrent bugs

- Knowing the types of concurrent bugs that actually occur in software will:
  - Help create better bug detection schemes
  - Inform the testing process software goes through
  - Provide information to program language designers
- Repeating concurrent bugs is difficult
- Testing is critical to find concurrency bugs

Concurrency bug types

- Data race: occurs when two conflicting accesses to one shared variable are executed without proper synchronisation, e.g. not protected by a common lock
- Deadlock: occurs when two or more operations circularly wait for each other to release the acquired resource (e.g. locks)
- Atomicity violation bugs: bugs which are caused by concurrent execution unexpectedly violating the atomicity of a certain code region
- Order violation bugs: bugs that don't follow the programmers intended order

Data race bug example



a[2] is updated in the second thread before a[1] uses it in the first thread. Wrong a[1] gets generated.

Preventing data races

- Using locks or atomic operations on shared variables

Orig. Code with data race

```
public class Counter {
   int counter;

   public void increment() {
      counter++;
   }
}
```

Data race on *counter shared variable.*
counter++ is a combination of 3 operations: reading the value, incrementing and writing the updated value.

Using Locks to prevent data race

```
public class SafeCounterWithLock {
   private volatile int counter;

   public synchronized void increment() {
      counter++;
   }
}
```

Volatile keyword for reference visibility among threads.
The **synchronized** keyword acts as a lock and ensures that only one thread can enter the method at one time.

- Use atomic shared variables
  - Better than locks because locks will do it for the entire region of course (slower)

The most commonly used atomic variable classes in Java are **AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference**. These classes represent an int, long, boolean and object reference respectively which can be atomically updated. Methods exposed by these classes are *get(), set(), lazySet(), compareAndSet()*.

Using Atomics to prevent data race

```
public class SafeCounterWithAtomic {
   private final AtomicInteger counter =
                  new AtomicInteger(0);
   public int getValue() {
      return counter.get();
   }
   public void increment() {
         while(true) {
         int existingValue = getValue();
         int newValue = existingValue + 1;
         if(counter.compareAndSet
               (existingValue, newValue)) {
            return;
         }
      }
   }
}
```

Deadlock

- When two threads are waiting on each other for a particular resource
- Example

Figure - 1

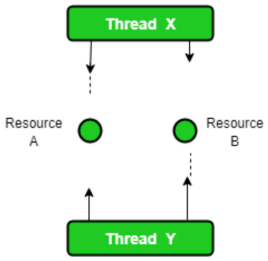
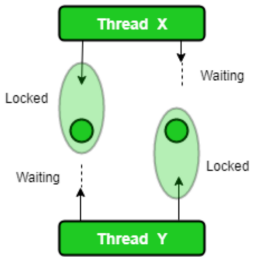


Figure - 2



Deadlock

thread 1	thread 2
void f1()	void f2()
{	{
get(A);	get(B);
get(B);	get(A);
release(B);	release(A);
release(A);	release(B);
}	}

Deadlock prevention

- Lock ordering
  - Deadlock occurs when multiple threads need the same locks ut obtain them in different order
  - If you make sure that all locks are always taken in the same order by any thread, deadlocks cannot occur
- Lock timeout
  - Put a timeout on lock attempts
  - A thread trying to obtain a lock will only try for so long before giving up
  - If a thread does not succeed in taking all necessary locks within the given timeout, it will backup, free all locks taken, wait for a random amount of time and then retry
  - The random amount of time waited serves to give other threads trying to take the same locks a chance to take all locks, and thus let the application continue running without locking

Atomicity violation

- When you expect a section of code to execute sequentially but it doesn't
- Example
  - Atomicity violation detection techniques check if an observed execution is equivalent to any serial execution

Case 1

```
thread-1                thread-2
deposit(int val){       deposit(int val){
   int tmp = bal;          int tmp = bal;
   tmp = tmp + val;        tmp = tmp + val;
   bal = tmp;              bal = tmp;
}                       }
```

Case 2

```
thread-1                thread-2
deposit(int val){       deposit(int val){
   synchronized(o){        synchronized(o){
     int tmp = bal;          int tmp = bal;
     tmp = tmp + val;        tmp = tmp + val;
   }                       }
   synchronized(o){        synchronized(o){
     bal = tmp;              bal = tmp;
   }                       }
}                       }
```

  - The programmer may be required to enforce sequential execution of operations as a whole to avoid atomicity violations

Order violation bug in Mozilla

- When you expect statements to execute in a certain order but it doesn't do so
- Order violation bugs can be prevented using synchronisation variables and locks