# SWEN90006 Notes

# Class Notes

# Week 1

## Software Testing

consists of the **dynamic verification** that a program provides **expected behavior**s on a **finite set** of **test cases, suitably selected** from the usually infinite execution domain. (IEEE SWEBOK)

- Dynamic: **Program can be executable**
- Expected Behavior: **Behaviour matches the requirements**
- Finite Set: **Fix number of inputs** that results in outputs
- Suitably Selected: Some method or technique or strategy

It means executing a program in order to measure its attributes.

- Measuring a program's attributes means that we want to work out if the program fails in some way, work out its response time or through-put for certain data sets, its mean time to failure (MTTF), or the speed and accuracy with which users complete their designated tasks.

software testing means executing a program or its components in order to assure:

1. The correctness of software with respect to requirements or intent;
2. The performance of software under various conditions;
3. The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;
4. The security of software, that is, the absence of vulnerabilities and its robustness against different kinds of attack;
5. The usability of software under various conditions;
6. The reliability, availability, survivability or other dependability measures of software; or
7. Installability and other facets of a software release.

Testing is described as part of the process of **validation** and **verification**.

### Validation

- the process of evaluating a system or component during or at the end of the development process to determine if it **satisfies the requirements of the system**.
- In other words, validation aims to answer the question: Are we building the correct system?

### Verification

- the process of evaluating a system or component at the end of a phase to determine if it **satisfies the conditions imposed at the start of that phase**.

- In other words, verification aims to answer the question: Are we building the system correctly?

### Terminate

For terminating the program, execute it and examine the returned value. Even if it terminates we must still examine the array that was passed as input, because its value may change.

1. test the program or function by testing that the sequence of values that it produces is what we expect.

### Not terminate

A classic example of a non-terminating program is a control loop for an interactive program or an embedded system. Control loops effectively execute until the system is shutdown.

1. non-terminating program may:
    - generate observable outputs; or
    - not generate any observable outputs at all.
2. need to examine the internal state somehow.

### Note:

1. Not all attributes of a program can be quantified.
    a. Some attributes, like reliability, performance measures, and availability are straightforward to measure.
    b. Others, such as usability or safety must be estimated using the engineer's judgment using data gathered from other sources.
        i. For example, we may estimate the safety of a computer control system for automated braking from the reliability of the braking computers and their software.
2. Inspections, walk-throughs and audits as part of the V&V process but not part of testing.

## Testing Purpose

Software testing methods are used to evaluate and assure that a program meets all of its requirements, both functional and non-functional.
**Major focus** will be on **functional correctness** of the software.
The aim of most testing methods is to systematically and actively find these discrepancies in the program.

The purpose of testing programs is to demonstrate

- Does the software produce the outputs consistent with the software specification, given the inputs?

- Testing directed at revealing as many defects in the software under test as possible given the available testing resources.
- Discrepancy between an implementation and its specification.
    - That means there is a fault in a program
- Basis of different testing strategies
- To show that our software is correct.
- To find ALL the faults in our software.
- The prove that our software meets its specification.

Testing and debugging are **NOT** the same activity.

### Debugging

1. The purpose of debugging is to locate faults and fix them.
    a. Debugging is the activity of:
        i. determining the exact nature and location of a suspected fault within the program; and
        ii. fixing that fault. Usually, debugging begins with some indication of the existence of a fault.
    b. The aim of debugging is to locate the cause of these faults, and remove or repair them.

### Testing

The aim of testing is to demonstrate that there are faults in a program.

### Program Proving(Formal Program Verification)

The aim of program proving is to show that the program does not contain any faults.

- The problem with program proving is that most programmers and quality assurance people are not equipped with the necessary skills to prove programs correct.

## Failure

**Failure happens as a result of a fault**. A failure occurs when there is a deviation of the observed behavior of a program, or a system, from its specification. A failure can also occur if the observed behavior of a system, or program, deviates from its intended behavior which may not be captured in any specification.
**In testing we can only ever detect failures.**

### Example

int square(int x) { return x*2; }

1. Executing square(3) results in 6, final result 6 is a failure.

# Fault

A fault is **an incorrect step, process, or data definition** in a computer program. In systems it can be more complicated and may be the result of an incorrect design step or a problem in manufacture.

## Example

int square(int x) { return x*2; }

1. Executing square(3) results in 6, code return x*2 is the fault part.

# Errors

Error is **an incorrect internal state** that is the result of some fault. **An error may not result in a failure** – it is possible that an internal state is incorrect but that it does not affect the output.

## Example

int square(int x) { return x*2; }

1. Executing square(3) results in 6, code x*2 is the error.

Fault results failure or error

**Failures and errors are the result of faults** – a fault in a program can trigger a failure and/or an error under the right circumstances.

In normal language, software faults are usually referred to as “bugs”, but the term “bug” is ambiguous and can mean to faults, failures, or errors.

# Coincidental Correctness

Correct behavior(Output) even though the function is implemented incorrectly.

## Example

int square(int x) { return x*2; }

2. Executing square(2) results in 4.

# Three steps

1. Detect system failures by choosing test inputs carefully;
2. Determine the faults leading to the failures detected;
3. Repair and remove the faults leading to the failures detected; and test the system or software component again.

This process is itself error-prone. We must not only guard against errors that can be made at steps (2) and (3) but also note that new faults can be introduced at step (3). It is important to realize here that steps (2) and (3) must be subject to the same quality assurance activities that one may use to develop code.

# Test Cases

Testing comes down to the selecting and executing test cases.
A collection of test cases is a **test suite**.
A test case for a specific component consists of three essential pieces of information:

- a set of test inputs, or if the program under test is non-terminating, a set of sequences of test inputs;
- the expected results when the inputs are executed; and
- the execution conditions or execution environment in which the inputs are to be executed.

## Test Activities

1. Plan the test
2. Select Test cases
3. Execute test cases
4. Evaluate Tests
5. Report Tests

## Test Case Selection

Test case selection is typically performed at a high level to produce abstract test cases, and these are then refined into executable test cases.

### Two steps

1. Select the test inputs. This is typically performed using a test input select technique, which aims to achieve some form of coverage criterion.
2. Provide the expected behavior of every test input that is generated.
    a. Referred to as the oracle problem.
    b. In many cases, this oracle can be derived in a straightforward manner from the requirements of the program being tested.
    c. The Oracle problem is a difficult problem, and it is difficult to automate oracles or to assess their quality.
    d. Example:
        i. a test case that assesses performance of a system may be related to a specific requirement about performance in the requirements specification of that system.

## Test Execution

Execute the test inputs on the program-under-test, and record the actual behavior of the software.

- Test execution is one step of the testing process that is generally able to be **automated**. This not only saves the tester time, but allows for regression testing to be performed at minimal cases.
- Example

- Record the output produced by a functional test input, or measure the time taken to execute a performance test input.

### Test Evaluation

Compare the actual behavior of the program under the test input with the expected behavior of the program under that test input, and determine whether the actual behavior satisfies the requirements.

- Test evaluation can generally be **automated**.
- Example
  - In the case of performance testing, determine whether the time taken to run a test is less than the required threshold.

### Test Reporting

Report the outcome of the testing. This report may be returned to developers so they can fix the faults that relate to the failures, or it may be to a manager to inform them that this stage of the testing process is complete.

- Can be automated, **depending on the requirements** of the report.

## Test Planning

A test plan allows

1. review of the adequacy and feasibility of the testing process.
2. review of the quality of the test cases.
3. providing support for maintenance.

### Minimum requirement

be written for every artifact being tested at every level. Also, contains:

1. Purpose
   a. Identifies what artifact is being tested, and for what purpose;
      i. Eg. For functional correctness, performance, security, etc;
2. Assumptions
   a. Any assumptions made about the program being tested;
3. Strategy
   a. The strategy for test case selection;
      i. Blackbox; Whitebox; fuzzing, metamorphic etc.
4. Supporting artifacts
   a. A specification of the supporting artifacts.
      i. eg. test stubs or drivers;
5. Test Cases
   a. A description of the abstract test cases, and how they were derived.

Other information can be included in a test plan, such as the estimate of the amount of resources required to perform the testing.

### Successful test case

A test is successful if it fails.

- Any test that doesn't find a fault is a waste.
- Anything that does not reveal a problem is not valuable.
- We must consider programs as sick patients – they contain faults whether we want them to or not.

## Testability

Testability is composed of controllability and observability. They are difficult to measure, but must be considered during the design of software.
Large impact on

1. the amount of testing that is performed.
2. the amount of time that must be spent on the testing process to achieve certain test requirements.

### Controllability

the degree to which a tester can provide test inputs to the software.

### Observability

the degree to which a tester can observe the behavior of a software artifact.
Example: its outputs and its effect on its environment.

Software with a user interface is generally more difficult to control and observe.

- Test automation software exists to record and playback test cases for graphical user interfaces, however, the first run of the tests must be performed manually, and expected outputs observed manually.
- In addition, the record and playback is often unreliable due to the low observability and controllability.

Embedded software is generally less controllable and observable than software with user interfaces.

- A piece of embedded software that receives inputs from sensors and produces outputs to actuators is likely to be difficult to monitor in such an environment — typically much more difficult than via other software or via a keyboard and screen.
- While the embedded software may be able to be extracted from its environment and tested as a stand-alone component, **testing software in its production environment is still necessary**.

# The psychology of Software Testing

## The purpose of software testing

Testing shows the presence of faults, but not their absence, is important.

**Psychology facts**

1. Software is complex.
2. Every piece of software has faults.
3. Every program has an infinite number of possible inputs.
4. We can't test all except some trivial ones.
5. No matter how many tests were run on this program, there is no guarantee that the very next test will not fail.
6. we can't prove a program is correct with testing.
7. Any program we test will have faults, and will continue after we finish testing.

Testing is NOT proof. Testing is a destructive process. We try to break our program.

## Regression test

Keep tests and run them later after debugging to make sure we do not introduce any new faults.

# Software Testing Psychology

## Principle 1

A necessary part of a test case is a definition of the expected output or result.

- Before we run the test, we must also know what the expected output should be. It is not sufficient to run the test, see the output, and only then decide whether that output is correct.

## Principle 2

A programmer should avoid attempting to test his or her own program.

- A programmer should avoid being the only person to test his or her own program.
- Reason
    a. If a programmer missed important things when coding, then it is quite likely they will also not think of these during testing. Such as failing to consider a null pointer, or failing to check a divide by zero. However, another person is less likely to make exactly the same mistakes. So, this **duplication helps to find these types of 'oversight' faults**.
    b. If a programmer misunderstands the specification they are programming to (e.g. they misunderstand a user requirement), they will **implement incorrect code**. When it comes to testing, they will still misunderstand the specification, and will therefore write an incorrect test using this misunderstanding. Both the code and the test are wrong, and in the same way. To the test will pass. However, **another**

**person brings an opportunity to interpret the specification correctly**. Of course, they may interpret it incorrectly in the same way, but it is less likely that both people will do this rather than just one.

c. Recall that testing is a destructive process. Someone testing their own code will struggle to switch from the constructive process of coding to the destructive process of testing if the code is their own. **Psychologically, they will semi-consciously try to avoid testing parts of the code they think are faulty.**

### Principle 8

The probability of the existence of more errors in a section of a program is proportional to the numbers already found in that section.

Reason for Faults in clusters

1. complex bits of code are harder to get right.
2. Some parts of the code are written hurriedly due to time constraints.
3. Some software engineers are not as good as others, so their parts will be more faulty.

It means that as we test, we find many faults in one part of a program and fix them, and find comparatively fewer in another part and fix them. We should then invest our time in the more faulty regions. **Best investment is made in these error/fault-prone sections.**

## Input Domains

Need to work out the sets of values making up the input domain.
Systematic testing aims to cover the full input domain with test cases so that we have assurance – confidence in the result – that we have not missed testing a relevant subset of inputs.

### Constructed from

- Inputs passed in as parameters;
- Inputs entered by the user via the program interface;
- Inputs that are read in from files;
- Inputs that are constants and precomputed values;
- Aspects of the global system state including:
    - Variables and data structures shared between programs or components;
    - Operating system variables and data structures, for example, the state of the scheduler or the process stack;
    - The state of files in the file system;
    - Saved data from interrupts and interrupt handlers.

### Variables can be

- atomic data such as integers and floating point numbers;

- structured data such as linked lists, files or trees;
- a reference or a value parameter as in the declaration of a function;
- constants declared in an enclosing scope of the function under test.

### Two sources about gather information

1. the software requirements and design specifications;
2. the external variables of the program you are testing.

## Specification-based Testing Strategies

COTS stands for Commercial Off The Shelf.

The terms “black-box testing” and “white-box testing” are **becoming increasingly blurred**.

- Many of the white-box testing techniques that have been used on programs, such as control-flow analysis and mutation analysis, are now being applied to the specifications of programs. That is, given a formal specification of a program, white-box testing techniques are being used on that specification to derive test cases for the program under test. Such an approach is clearly black-box, because test cases are selected from the specified behaviour rather than the program, however, the techniques come from the theory of white-box testing.

### Black-box Testing

Where test cases are derived from the **functional specification** of the system.

- Goal is to check whether specified functionality is available and working correctly.
- Black-box test case selection can be done without any reference to the program design or the program code.
- Black-Box test cases test **only the functionality and features of the program** but not any of its internal operations.
- Good at testing for missing functions and program behavior that deviates from the specification.
    - They are ideal for evaluating products that you intend to use in a system such as COTS [6] products and third party software (including open source software).
- Knowledge sources
    - Requirements documents
    - Specifications
    - Domain Knowledge
    - Defect analysis data
- Method
    - Equivalence class partitioning
    - Boundary value analysis
    - State transition testing
    - Cause & Effect graphing

- Error Guessing

**Advantages**
It can be done before the implementation of a program. This means that black-box test cases can **help in getting the design and coding correct** to the specification, because it will be compared with specification.

**Disadvantages**
black-box test cases cannot detect additional functions or features that have been added to the code. This is especially important for systems that are safety critical (additional code may interfere with the safety of the system) or need to be secure (additional code may be used to break security). **Unexpected functionality can not be revealed by black box techniques.**

## White-box Testing

Where test cases are derived from the internal design specifications or actual code(**implementation**) for the program (sometimes referred to as glass-box).

- Goal is to check whether the implementation is working correctly(no dead code, maintainable), useful for debugging.
- Selected using requirements and design specifications and the code of the program under test. This means that the **testing team needs access to the internal designs and code for the program**.
- Knowledge sources
  - High-level design
  - Detailed design
  - Control flow
  - Graphs
  - Cyclomatic complexity
  - Data
- Method
  - Statement testing
  - Branch testing
  - Patch testing
  - Data flow testing
  - Mutation testing
  - Loop testing

**Advantages**
It tests the internal details of the code and tries to check all the paths that a program can execute to determine if a problem occurs. As a result of this white-box test cases **can check any additional code that has been implemented but not specified**.

**Disadvantages**
Must wait until after designing and coding the program under test in order to select test cases. In addition, **Missing functionality can not be revealed by using white-box testing**.

# Error Guessing

Error guessing is an **ad-hoc** approach based on intuition and experience. The idea is to identify test cases that are considered likely to expose errors.

- The general technique is to make a list, or better a taxonomy (a hierarchy), of possible errors or error-prone situations and then develop test cases based on the list.
- The idea is to document common error-prone or error-causing situations and create a defect history. We use the defect history to derive test cases for new programs or systems.
- Not a testing technique that can be assessed for usefulness or effectiveness.
    - Because it relies heavily on the person doing the guessing.
    - It takes advantage of the fact that programmers and testers both generally have extensive experience in locating and fixing the kinds of faults that are introduced into programs, and can use their knowledge to guess the test inputs that are likely to uncover faults for specific types of data and algorithm.
- ad-hoc, not systematic. The rest of the techniques described in these notes are systematic, and can therefore be used more effectively as quality assurance activities.

## Possible sources for defect history

1. **The Testing History of Previous Programs** — develop a list of faults detected in previous programs together with their frequency;
2. **The Results of Code Reviews and Inspections** — inspections are not the same as code reviews because they require much more detailed defect tracking than code reviews; use the data from inspections to create
3. From experience
    a. Depends on the developers and testers intuition and experience.
    b. Null Checks
    c. Array Bounds
    d. Arithmetic Errors (divide by zero)

## Common faults

1. Test cases for empty or null strings, array bounds and array arithmetic expressions (such as attempting to divide by zero)
2. Blank values, control characters, or null characters in strings.

## Testing Laws

Not really laws per se, but rules of thumb that can be useful for software testing.

- Dijkstra's Law: Testing can only be used to show the presence of errors, but never the absence of errors.
- Hetzel-Myers Law: A combination of different V&V methods out-performs and single method alone.
- Weinberg's Law: A developer is unsuited to test their own code.
- Pareto-Zipf principle: Approximately 80% of the errors are found in 20% of the code.
- Gutjar's Hypothesis: Partition testing, that is, methods that partition the input domain or the program and test according to those partitions, is better than random testing.
- Weyuker's Hypothesis: The adequacy of a test suite for coverage criterion can only be defined intuitively.

# Week 2

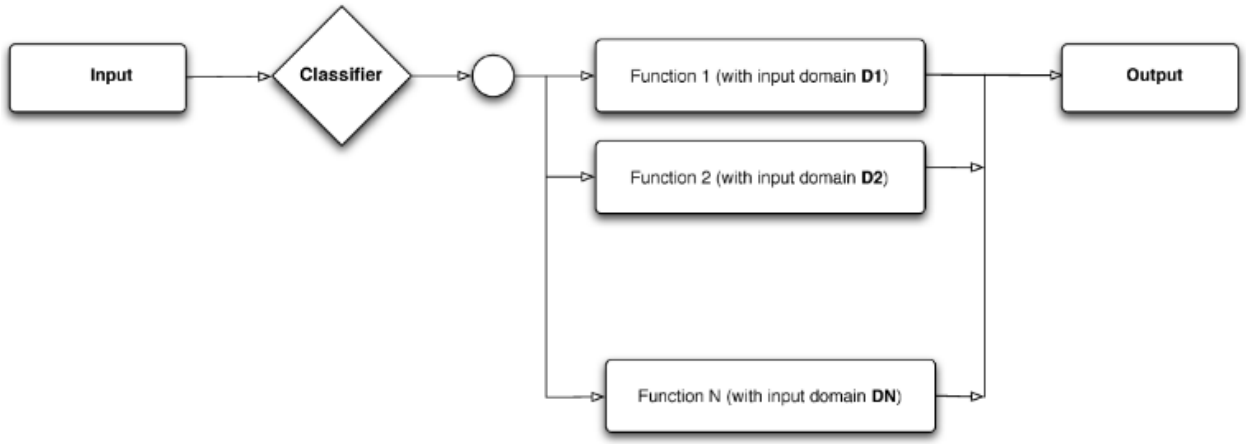
## Input Partitioning

A systematic method for identifying interesting input conditions to be tested.
The aim of input partitioning is to derive test inputs that exercise each function of the program at least once.
Using input partitioning, programs are considered as the composition of an input classifier, that classifies the input domain into one of a number of different classes where each input class computes one function of the overall program.

To make our use of the term input condition consistent with other literature we will **assume** that an input condition can be applied to a set of values of a specific input variable, or a set of input variables as well.
The assumption behind equivalence partitioning is that all members of the class behave in the same way with respect to failures

- Choosing one member of an equivalence class has the same likelihood of detecting a failure as any other member of the equivalence class.

This assumption is **not correct**, but partitioning the input domain into equivalence classes is a valuable technique for testing when **combined with other techniques**.




One-to-one correspondence between the input domains and the functions computed by a program.
Each of the functions computed by a program occurs along a program path.

- a path in the program that executes a sequence of statements for computing the function.

In domain testing, an input domain is the subset of all inputs that will trigger a specific program function to be computed along a specific program path.

## Two possible types of faults

**Computation faults**
where the correct path is chosen but an incorrect computation occurs along that path.

**Domain faults**
where the computation is correct for each path but an incorrect path is chosen.

**The possible causes for an incorrect path**

1. the incorrect path is executed for the input domain
2. the decisions that make up the path selection may contain a fault
3. the correct path (or a fragment of a path necessary for the computation) may simply be missing.

## Equivalence Classes

In functional testing, derive equivalence classes systematically and test inputs that are more likely to find faults than by using random testing.

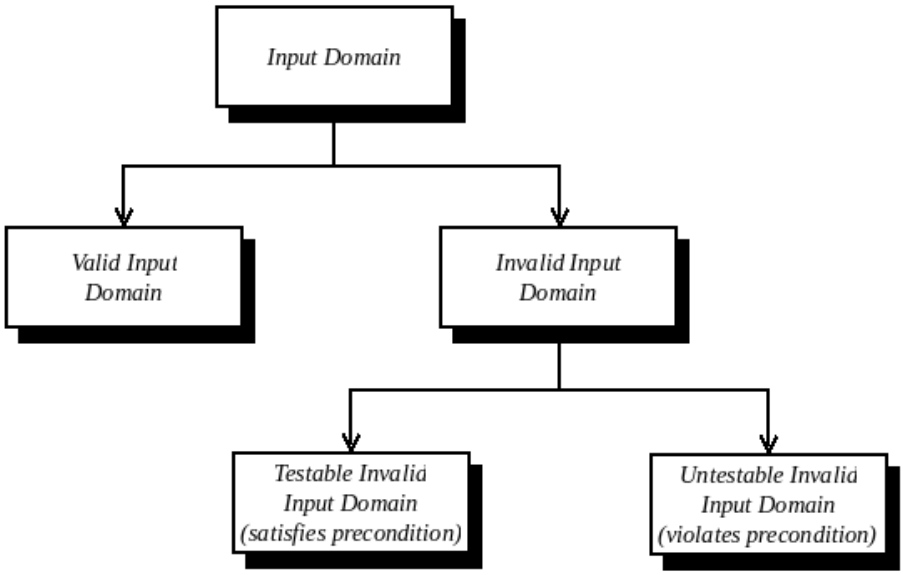
- An equivalence class is a set of values from the input domain that are considered to be as likely as each other to produce failures for a program.
- A single test input taken from an equivalence class is representative of all of the inputs in that class.

    - Complex explanation: For an equivalence class, each element in that equivalence class should execute the same statements in a program in the same order, but just with different values.
- Each equivalence class is used to represent certain input conditions on the input domain.

Equivalence partitioning tries to break up input domains into sets of valid and invalid inputs based on these input conditions.

**Input condition**
A predicate on the values of a single variable in the input domain.

- When we need to consider the values of more than one variable in the input domain, we refer to this as the **combination of input conditions**.




**Valid input**
An element of the input domain that is the value expected from the program specification.

- A non-error value according to the program specification.

**Invalid input**
an input to a program is an element of the input domain that is not expected or an error value that as given by the program specification.

- an invalid input is a value in the input domain that is not a valid input.

Two sub class

1. Testable
    a. An invalid input that does not violate the precondition. Typically, these refer to inputs that return error codes or throw exceptions.
2. Non-testable
    a. Input violates the precondition of a program.

i. A precondition is a condition on the input variables of a program that is assumed to hold when that program is executed.
ii. The behavior of the program for any input that violates the precondition is undefined.
iii. If the behavior is undefined, then we do not test for it.

**Two key properties**

There are two key properties equivalence classes for software testing. If we have $n$ equivalence classes $EC_1, \ldots, EC_n$ for an input domain, $ID$, the following two properties must hold:

- for any two equivalence classes, $EC_i$ and $EC_j$, such that $i \neq j$, $EC_i \cap EC_j = \emptyset$; that is, $EC_i$ and $EC_j$ are *disjoint*; and
- $\bigcup\{EC_1, \ldots, EC_n\} = ID$; that is, the testable input domain is covered.

In conclusion, the equivalence classes are disjoint and they cover the input domain. For any value in the input domain, that input belongs to exactly one equivalence class.

## Two systematic techniques for partitioning

**Domain Testing**

A **white-box testing technique** relies on the underlying structure of the program to determine which inputs execute the same program statements, determining the actual equivalence classes of a program.

Domain testing partitions the input of a program **using the program structure itself.** The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

Select test cases

1. The input space is first partitioned into a set of mutually exclusive equivalence classes. Each equivalence class corresponds to a program path.
    a. test cases are selected from
        i. the domain boundaries
        ii. points close to the domain boundaries.

Standard Terminology

- Relational Expressions
    - A rel B
        - rel means one of $>, <, \geq, \leq, =$ and $\neq$.
- Predicates

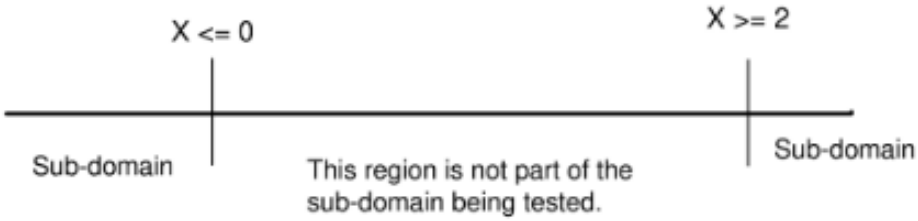
    - Points in a program where the control flow can diverge.
- Simple Predicates
    - Single relational expressions.
- Compound Predicates
    - predicates composed of several simple predicates combined by boolean operators. AND (∧), OR (∨) and NOT (¬).
- Linear Simple Predicates
    - **Linear Simple Predicates**: are simple predicates of the form $(a_1x_1 + \ldots + a_nx_n)\ \ rel\ \ k$, where $rel$ is a relational operator, $x_1, \ldots, x_n$ and the $a_i$s and $k$ are constants.
- Path Condition
    - The condition that must be satisfied by the input data for that path to be executed.

**Domain testing makes an assumption that programs are linearly domained.** Linear borders require fewer test points to check and can be checked with more certainty. If a program violates this assumption then testing using this strategy may not be as productive at finding faults as would have been the case if the program did satisfy the assumptions.

- Because solving non-linear equalities and inequalities is considerably more difficult than solving linear equalities and inequalities.
- In the non-linear case a point may lie in a domain but be computationally too expensive to check accurately.

Compound Predicates

- A compound predicate with a logical conjunction (AND operator) is straight-forward because it often defines a convex domain where all of the points lie within a boundary.
- A compound predicate with a logical disjunction (OR operator) can create disjointed boundaries. Eg. x<= 0 v x >=2




To overcome this problem, domain testing treats compound predicates as different paths.
In the example, the cases are treated as two separate functions. Each compound predicate is broken into disjunctive normal form; A disjunction of one or more simple predicates.

$$\neg(A \vee B) \ \equiv \ \neg A \wedge \neg B$$

we can reduce the expression $\neg(x < 0 \vee x > 2)$ to $x > 0 \wedge x < 2$.

Loops in Program
Root cause

- If each program path corresponds to a function, then for some programs containing loops, we have an infinite number of paths, and therefore, an infinite number of equivalence classes. Selecting test inputs for each of these classes is clearly impossible, so we need a technique for making this finite.

Solution
A typical work around is to apply the zero-to-many rule. The zero-to-many rule states that the minimum number of times a loop can execute is zero, so this is one path. Similarly, some loops have an upper bound, so treat this as a path. A general guideline for loops is to select test inputs that execute the loop:

- zero times – so that we can test paths that do not execute the loop;
- once – to test that the loop can be entered and that the results for a single iteration are correct;
- twice – to test that the results remain correct between different iterations of the loop;
- N times (greater than 2) – to test that an arbitrary number of iterations returns the correct results;
- N+1 times to test that after an arbitrary number of iterations the results remain correct between iterations.

In the case that we know the upper bound of the loop, then set N+1 to be this upper bound. Therefore, we are partitioning the input domain based on the number of times that a loop executes. We need only to instantiate the number, N, when selecting the test inputs that satisfy the equivalence classes.

**Equivalence partitioning**
A **black-box testing technique** relies on the specified functionality of the program, determining the expected equivalence classes of a program from its functional requirements.
Equivalence partitioning partitions the input of a program **using the functional requirements of the program**. The test inputs are executed on the program, and the tester evaluates whether the output of the program is in the expected output domain.

- The aim is to minimize the number of test cases required to cover all of the equivalence classes that you have identified.
- Selecting just any value in an equivalence class may not be optimal for finding faults.Select more than one input from an equivalence class, which reduces the chances of coincidental correctness, but increases the size of our test suite.

Three Steps

1. Identify the initial equivalence classes (ECs);

a. **No clear cut** algorithm or method for choosing test inputs according to equivalence partitioning. Need to build up some judgment and intuition.
b. guidelines help to identify potential equivalence classes
    i. Range
        1. If an input condition specifies a range of values, identify one valid equivalence class for the set of values in the range, and two invalid equivalence classes; one for the set of values below the range and one for the set of values above the range.
        2. Example:
            a. Range from 1 - 99.
                i. Valid EC is 1 - 99
                ii. invalid EC are {x|x<1} && {x|x>99}
    ii. Discrete Sets
        1. If an input condition specifies a set of possible input values and each is handled differently, identify a valid equivalence class for each element of the set and one invalid equivalence class for the elements that are not in the set.
        2. Example
            a. Input select from a set of vehicle types {BUS,TRUCK,TAXI,MOTORCYCLE}
                i. Valid EC are single element in the set
                ii. Invalid EC are TRAILER or NON_VEHICLE.
    iii. Number of Inputs
        1. If the input condition specifies the number (say N) of valid inputs, define one valid equivalence class for the correct number of inputs and two invalid equivalence classes – one for values < N and one for values > N.
        2. Example
            a. Users should input 3 preferences.
                i. Valid EC for 3 preferences.
                ii. Invalid EC for < 3 and > 3.
    iv. Zero-One-Many
        1. If the input condition specifies that an input is a collection of items, and the collection can be of varying size
        2. Example
            a. a list or set of elements
                i. Valid EC for a collection of size 0, and a collection of size 1, and a collection of size.
        3. If with bounds
            a. a list or set of elements with maximum number N

i. Valid EC for a collection of size N,
ii. Invalid EC for < N and > N.

v. Must rule
    1. If an input condition specifies a “must be” situation, identify one valid equivalence class and one invalid equivalence class.
    2. Example
        a. Input be a int
            i. Valid EC for the first character is int.
            ii. Invalid EC for the first character is not int.

vi. Intuition/Experience/Catch-all
    1. If there is any reason to believe that elements in an equivalence class are handled in a different manner than each other by the program, then split the equivalence class into smaller equivalence classes.
    2. This is a sort of “default” catch statement, you can derive tests based on intuition or your understanding of the program and domain, where none of the other guidelines fit.

2. Identify overlapping equivalence classes, and eliminate them by making the overlapping part a new equivalence classes;
    a. Once these overlapping classes has been identified, they are eliminated by treating them as equivalence classes themselves.
    b. Sometimes end up with less equivalence classes, however, in other cases may end up with more.
3. Select one element from each equivalence class as the test input and work out the expected result for that test input.
    a. Any value from an equivalence class is identified to be as likely to produce a failure as any other value in that class, therefore any element of the class serves as a test input, and selecting any arbitrary element from the class is adequate.

**Test Template Trees**

A test template tree is an overview of an equivalence partitioning, **it is hierarchical**. When applied correctly, it provides a graphical overview of equivalence classes and their justification, as well as avoiding overlap.

Process

1. Start
    a. Start with the testable input domain, and aim to break this into smaller equivalence classes that make good tests.
2. Repeat

    a. At each step, choose an equivalence partitioning guideline to apply to one or more leaf nodes in the test template tree, breaking the leaf nodes into multiple new leaf nodes, each once more specific than its parent node, which is no longer a leaf node.
    b. When partition each leaf node, ensure the partitioning is
        i. disjoint — the partitions do not overlap
        ii. they partitions cover their parent
            1. if we combine the new partitions, that combination will be equivalent to their parent.
    c. These two properties are the same as listed in Section Equivalence Classes for equivalence classes. By avoiding creating overlap and by covering each parent, we ensure (inductively) that the resulting equivalence classes at the end of this entire process also do not overlap and their cover the root node (the input domain).
3. End
    a. The process ends when either
        i. there are no more sensible partitionings to apply
        ii. our tree grows too large and we can no longer manage the complexity
4. Result
    a. To construct the equivalence classes, we simply take each leaf node and derive the equivalence from that lead node to the root of the tree.
    b. The resulting equivalence classes do not overlap and they cover the entire input domain.
        i. This is because, at each partition, we ensure this property is held locally. This local property means that non-overlap and coverage also hold globally, which can be demonstrated inductively.
5. Mitigating Tree Explosion
    a. Generating partitions like this can result in trees that are prohibitively large. The problem is caused by the nature of the partitioning: if we apply a guideline to every leaf node of a tree, then the number of nodes grows exponentially with the depth of the tree.
    b. To mitigate the problem
        i. Test invalid, exceptional, and error cases only once
            1. Many test inputs involve invalid or exceptional cases, such as throwing an exception or returning an error codes. For these cases, aim to have only one node in the tree that corresponds to each trigger for these exceptional cases.
            2. Applying guidelines to exceptional cases will probably not (but not certainly not) provide good test inputs.

a. This is because most people use defensive programming, and therefore errors/exceptions are detected at the start of programs and the error/exception is thrown immediately.
b. As such, the remainder of the program is not executed.

3. So, if a node in a test template tree is designed to throw an exception due to an ill-formed input, creating child partitions that test several different inputs for 'normal case' functionality are unlikely to find any additional faults because the corresponding code will not be executed.

ii. Look at variable interactions

1. When applying a guideline, use experience and intuition and apply only to the current leaf nodes in which this partition is likely to involve some interaction between variables.
2. Example
    a. a system for recording loans from a library
        i. inputs such as date of loan, length of loan, name of book/DVD, author, etc.
        ii. Clear interaction between the date of the loan and the length of the loan, because these two variables determine the due date. However, the name of the book and the author (probably) do not affect the due date.
        iii. Then, when testing different values of loan length, it would be better to prioritize the nodes that focus on the loan date, rather than on the book name and author.
        iv. This reduces the number of partitions applied, and therefore mitigates the problem of explosion.

**Combining Partitions**

advocate looking at the variables of a program, and using test case selection techniques to derive equivalence classes.

- problems in which use multiple criteria, or one criteria over multiple variables. Removing the overlapping cases would generate a large number of test cases, many of which would be of little use.
- Ammann and Offutt call these **block combinations**
    - While the equivalence classes overlap, there are distinct blocks that are different to each other in some way.

Three different criteria to solve the problem.

All Combinations
The all combinations criterion specifies that every combination of each equivalence class between blocks must be used. This is analogous to the cross product of sets.

- The number of test cases is a product of the number of blocks and the number of equivalence classes in each block.
- In this case, it relates to the number of variables, and the number of equivalence classes for each variable. This is likely to be more test cases than necessary.

Pair-Wise combinations criterion
Each choice criterion specifies that just one test case must be chosen from each equivalence class. This is significantly weaker than the all combinations criterion, and does not consider combinations of values.

Each choice combinations criterion
The Pair-Wise combinations criterion aims to combine values, but not an exhaustive enumeration of all possible combinations.

- As the name suggests, an equivalence class from each block must be paired with every other equivalence class from all other blocks.
- One can generalize this to T-Wise Combinations, in which we require N number of combinations instead of pairs. It N is equal to the number of blocks, then this is equivalent to the all combinations criterion.

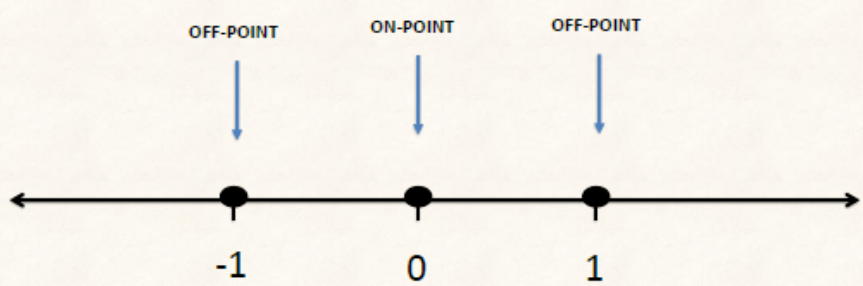
## Boundary-Value Analysis

Boundary-value analysis is both a refinement of input partitioning and an extension of it. Boundary conditions are **predicates** that apply directly on, above, and beneath the boundaries of input equivalence classes and output equivalence classes.

- Aims to select test cases to explore the boundary conditions of a program.
- Theory
    - if a programmer makes a mistake in the logic of the program, then those mistakes will often occur at the boundary between equivalence classes.
- For boundaries containing unordered types , e.g. Booleans or strings, choose one on point and one off point.
- **If ECs are incorrectly selected, the BCs will also be incorrect.**

**On-point**
the value that is on the boundary.

**Off-point**
the value closest to the boundary that flips the conditions




## Difference with Equivalence Partitioning

BVA

1. requires one or more test cases be selected from the edge of the equivalence class or close to the edge of the equivalence class.
2. Requires test cases be derived from the output conditions.

EP

1. Requires that any element in the equivalence class.
2. Usually consider the input domain only.

**Path Condition**

The condition must be satisfied by the input data for that path to be executed.

**Domain**

The set of input data satisfying a path condition.

**Domain Boundary**

The boundary of a domain typically corresponds to a simple predicate.

Closed Boundaries

The points on the boundary belonging to the domain, defined using an operator that contains an equality.

- On point will be a member of the equivalence class.
- Off point will fall outside of the equivalence class.

Open Boundaries

Not closed, defined using a strict inequality.

- On point will **not** be a member of the equivalence class.
- Off point will fall inside the equivalence class.

## Computational fault

Occurs during a computation in a program

- for example, an arithmetic computation or a string processing error.

## Boundary shift

when a predicate in a branch statement is incorrect, effectively ‘shifting’ the boundary away from its intended place.

- The tests on the boundaries can **detect both computational faults and boundary shifts**, while those away from the boundary defect only computational faults.
- Select one on point and one off point is not sufficient to detect shifts.

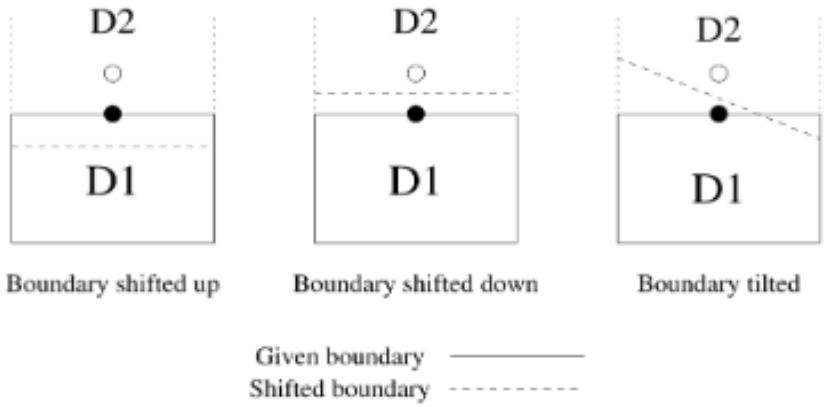
Root Cause

1. Unsure of the correct boundary for an input condition;
2. have incorrectly tested the boundary.

**Detect Boundary Shift in Inequalities**

Solution

For boundaries defined by an inequality, select two on points that are as far apart as possible (or reasonable for infinite domains), and one off point as close to the middle of the two on points as possible.




Boundary shift down

- The equivalence class is smaller than expected
    - because the on point is identified as being in the wrong domain.

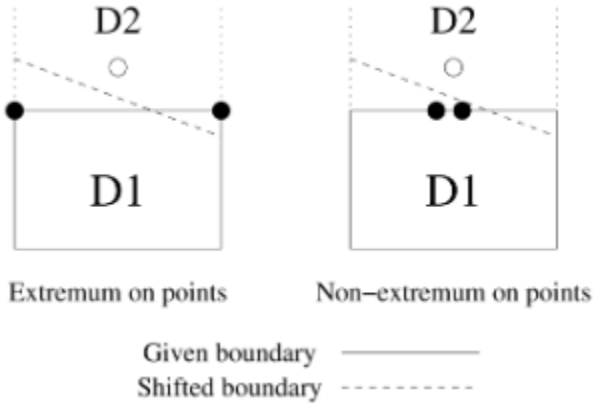
Boundary shift up

- Might not identify boundary shift.
- The equivalence class is larger than expected
    - The shift has to be great enough that the off point can detect it.
- If the off point is a certain distance away from the boundary, then only shifts at least as big as this distance can be detected by that off point.
- Solution
    - Choosing an off point that is close to the boundary is almost always likely to detect boundary shifts.

Boundary Tilted

The given boundary and the shifted boundary intersect at some point.

- Might not identify boundary shift.
- The on point falls within the boundary, and the off point falls outside of the boundary, therefore, these cases will not detect the boundary shift.
- If the values that can be selected have a minimum and a maximum (the extremum points), then **choose both of those as on points**.
    - Guarantees for any tilted boundary, at least one of the points must lie outside the shift boundary.




- The boundary has no extremums (or only a minimum or maximum but not both) due to it being infinite, then the further away the two on points are from each other, the higher risk of producing a failure.
- Once the two on points are selected, the best off point to select is one that is as close to the middle as possible between the two on points. This is known as the centroid.

**Detect Boundary Shift in Equalities**

Select the on point and off points (above and below).

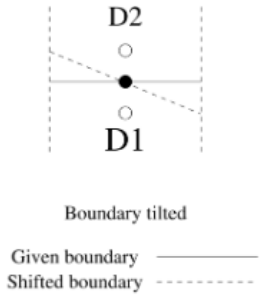
For a boundary shift up or down, these are enough to detect the shifts larger than the distance between the on point and off points.

**Equality boundaries are strictly equal.**

Solution

For boundaries defined by an equality, Select two on points, and two off points (one below and one above); and If the off point is of distance d, then only boundary shifts of magnitude greater than d can be detected.

The tilted shift will not be detected case




Solution

Choosing two on points, any on point other than the point at the intersection will detect the boundary shift.

**Boolean Boundaries**
The boundary is just the true/false value of the class.

# Week 3

## Coverage-Based Testing

- These techniques are useful only for selecting test inputs; not test cases. They are not useful for selecting test outputs.
- A white-box testing technique still requires a [test_oracles](test oracle), and therefore, a specification of the program behavior.
- Structural coverage criteria, such as control-flow and data-flow criteria, should be used to identify areas of the program that remain untested, and not as measure of the quality of the test suite. Structural test coverage is a **necessary but not sufficient method** for software testing.

Select better test cases to increase the power of each test case

1. Find better criteria for test case selection.
2. Find a testing scheme that produces additional information (information other than the output of the program under test) to use for program analysis.

## Control-Flow Testing

Control-flow strategies select test inputs to exercise paths in the control-flow graph. They select paths **based on the control information in the graph**, typically given by predicates in if statements and while loops.

Test inputs are selected to meet criteria for covering the graph (and therefore the code) with test cases in various ways. Examples of coverage criteria include:

- Path coverage;
- Branch coverage;
- Condition coverage;
- Statement coverage.

### Control-Flow Graphs(CFG)

The most common form of white-box testing, aims to understand the flow of control within a program.
CFG is a graphical representation of the control structure of a program and the possible paths along which the program may execute.

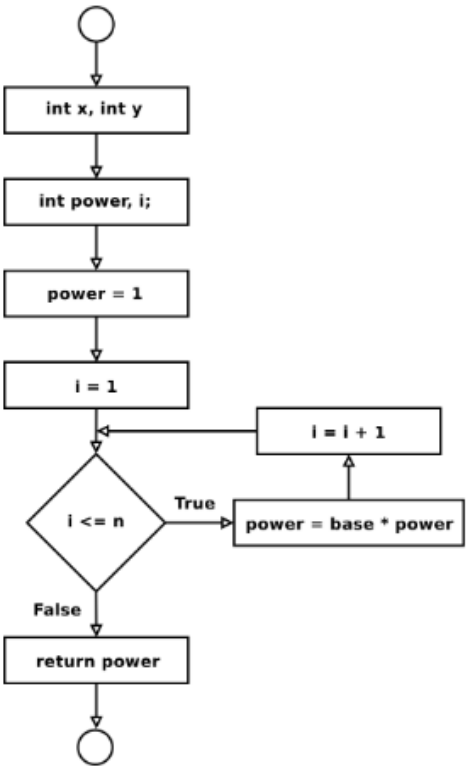
- statements are represented by square boxes.
- branches are represented by diamond boxes.

A control-flow graph is a graph g = (V,E) with

1. A vertex in the CFG represents a program statement;
2. An edge in the CFG represents the ability of a program to flow from its current statement to the statement at the other end of the edge;
3. If an edge is associated with a conditional statement, label the edge with the conditional value, either true or false.

Example

```
int power(int base, int n)
{
   int power, i;
   power = 1;
   for (i = 1; i <= n; i ++) {
      power = base * power;
   }
   return power;
}
```




Execution path

Or path, a sequence of nodes in the control flo ınd ends at the exit node.

Branch

Or decision, a point in the program where the flow of control can diverge.

- For example, if-then-else statements and switch statements cause branches in the control flow graph.

Condition

A simple atomic predicate or simple relational expression occurring within a branch. Conditions do not contain and (in C &&), or (in C ||) and not (in C !) operators.

Feasible path

There is at least one input in the input domain that can force the program to execute the path. Otherwise the path is an infeasible path and no test case can force the program to execute that path.

Common coverage criteria
Statement coverage
or node coverage. Every statement of the program should be exercised at least once.

Branch coverage
or decision coverage. Every possible alternative in a branch (or decision) of the program should be exercised at least once.

- For if statements this means that the branch **must be made to take on the values true and false**, even if the result of this branch does nothing.
- For an if statement with no else, the **false case must still be executed**.

Condition coverage
Each condition in a branch is made to evaluate to true and false at least once.

- For example, in the branch if (a and b), where a and b are both conditions, must run both
  - a evaluates to true at least once and false at least once
  - b evaluates to true at least once and false at least once.
  - a and b evaluates to true at least once and false at least once.

Multiple-condition coverage
All possible combinations of condition outcomes within each branch should be exercised at least once.

- For example, in the branch if (a and b), where a and b are both conditions, we must execute the following four conditions:
- For n conditions: 2^n number of objectives to satisfy.
  - 

| a | b | a and b |
|---|---|---|
| true | true | true |
| false | true | false |
| true | false | false |
| false | false | false |

Path coverage
Every execution path of the program should be exercised at least once.

- **Impossible for graphs containing loops**.
  - Use apply the zero-to-many rule, which states the minimum number of times a loop can execute is zero, so this is a boundary condition. Similarly, some loops have an upper bound, so treat this as a boundary.
- General guideline for loops

    - zero times – test paths that do not execute the loop;
    - once – to test that the loop can be entered and that the results for a single iteration are “correct”;
    - twice – to test that the results remain “correct” between different iterations of the loop;
    - N times (greater than 2) – to test that an arbitrary number of iterations returns the “correct” results; And N+1 times to test that after an arbitrary number of iterations the results remain “correct” between iterations.

Measuring Coverage

- A coverage score is defined as the number of test objects met divided by the number of total test objectives.
- The test objectives measured are relative to the particular criterion.
- The idea is to generate a test suite using some method (generally a black-box method), and to then measure the coverage of that test suite.
    - Test objectives that are not covered can then be added using the technique described in this section.
    - Such an approach is **cheaper** than deriving tests from the control-flow graph directly, due to the fact that a reasonable set of black-box tests will generally achieve a high coverage initially.

# Week 4

## Data-flow Testing

data-flow strategies select inputs to exercise paths **based on the flow of data between variables**.

- The aim in data-flow based testing methods is to select test cases that traverse paths from nodes that define variables to nodes that use those variables, and ultimately to nodes that undefine those variables.
- Eg. Between the definition of a variable, such as assigning a value to a variable, and the use of that variable in the program.

Test inputs are selected to cover the graph (and therefore the code) with test cases. Examples of coverage criteria include:

- Execute every definition;
- Execute every use;
- Execute every path between every definition and every use.

### Data-flow analysis

provides information about the creation and use of data definitions in a program.

Can be used to

- detect many simple programming or logical faults in the program;
- provide testers with dependencies between the definition and use of variables;
- provide a set of criteria to complement coverage based testing.
- Aim of data-flow analysis is to trace through the program's control-flow graph and detect data-flow anomalies. data-flow anomalies indicate the possibility of program faults.
- Does not tell what the fault actually is; it just tells that there is potentially something wrong.

**Static Data-Flow Analysis**

In the simplest form of data-flow analysis, a programming language statement may act on a variable in 3 different ways.

Defined(d)

A statement defines a variable by assigning a value to the variable.

- For example if the program variable x has been declared, then the statements x = 5 and scan(x) in C both defined the variable x, but the statement x = 3 * y only defines x if y is defined.

Reference (r)

A statement makes a reference to a variable by reading a value from the variable.

Undefine (u)

A statement undefines a variable whenever the value of the variable becomes unknown.

- For example, the scope of a local variable ends.

u-r anomaly

occurs when an undefined variable is referenced.

- Most commonly u-r anomalies occur when a variable is referenced without it having been assigned a value first, it is uninitialised.

d-u anomaly

occurs when a defined variable has not been referenced before it becomes undefined.

- Usually indicates that the wrong variable has been defined or undefined.

d-d anomaly

The same variable is defined twice causing a hole in the scope of the first definition of the variable.

- Usually occurs because of misspelling or because variables have been imported from another module.

Common programming mistakes can be detected

1. typing errors
2. uninitialised variables
3. misspelling of names
4. misplacing of statements
5. incorrect parameters
6. incorrect pointer references

**Dynamic Data-Flow Analysis**

- A C-use of a variable is a computation use of a variable.
    - for example, y = x*2;
- A P-use of a variable is a predicate use.
    - for example, if (x<2).
- Let dn(x) denote a variable x that is assigned or initialized to a value at node (statement) n ( **Definition**).
- Let un(x) denote a variable x that is used, or referenced, at node (statement) n (**Use**).
- Let cn(x) denote a computational usage of the variable x at the node n (**Computational Use**).
- Let pn(x) denote a predicate usage of the variable x at the node n **(Predicate Use**).
- Let kn(x) denote a variable x that is killed, or undefined, at a node (statement) n (**Kill**).

1. A **definition clear path** p with respect to a variable x is a sub-path of the control-flow graph where x is defined in the first node of the path p, and is not defined or killed in any of the remaining nodes in p.
2. A loop-free path segment is a sub-path p of the control-flow graph in which each node is visited at most once.
3. A definition dm(x) reaches a use un(x) if and only if there is a sub-path p that is definition clear (with respect to x, and for which m is the head element, and n is the final element.

**Forms of data-flow graph coverage criteria**

All-Defs

For the All-Defs criterion requires that there is some definition-clear sub-path from all definitions of a variable to a single use of that variable.

All-Uses

The All-Uses criteria requires some definition-clear sub-path from all definitions of a variable to all uses reached by that definition.

All-Du-Paths
DU stands for definition-use. The All-DU-Paths criterion requires that a test set traverse all definition-clear sub-paths that are cycle-free or simple-cycles from all definitions to all uses reached by that definition, and every successor node of that use.

**Additional data-flow test input selection criteria**
All-C-Uses, Some-P-Uses
The All-C-Uses, Some-P-Uses criteria requires a test set to traverse some definition-clear sub-path from each definition to each C-Use reached by that definition.

- If no C-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one P-Use reached by that definition.

All-P-Uses, Some-C-Uses
The All-P-Uses, Some-C-Uses requires that a test set to traverse some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use.

- If no P-Uses are reached by a definition, then some definition-clear sub-path from that definition to at least one C-Use reached by that definition.

All-P-Uses
Some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use.

# Mutation Analysis

A technique for measuring the effectiveness of test suites, with the **side effect** that test cases are created and added to that test suite.
**Based on seeding faults in a program**, and then assessing whether or not that fault is detected by the test suite. If not, then that test suite is inadequate because the fault was not detected, and a new test case must be added to find that fault.

- Using specially designed operators, many copies of the program are created, each one with a fault, with the aim of the test suite killing all of these.
- Given a test input for that program, and a mutant, say that the test case kills the mutant if and only if the output of the program and the mutant differ for the test inputs.
- Mutants are syntactic changes to a program, so mutant operators are dependent on the syntax of programming languages. Therefore, the mutant operators of one programming language may not necessarily apply to other languages.
    - Mutant operators **must** produce mutants that are syntactically valid.
    - Even if a language is dynamically interpreted, a good collection of mutant operators will ensure that any statically invalid programs are killed.

## Coupling Effect

The coupling effect states a test case that distinguishes a small fault in a program by identifying unexpected behavior is so sensitive that it will distinguish more complex faults; that is, complex faults in programs are coupled with simple faults.

- The coupling effect **has not been proven**, and in fact, **can not be proven**.
- The tester would aim to find a test case that does kill the mutant, and add this to the test suite.
- If test suite S kills all of the mutants that T kills, plus some additional mutants, then S subsumes T.

### Systematic Mutation Analysis

Mutation analysis is **only** effective if applied systematically. This is done using mutant operators.
mutant operator
a transformation rule that, given a program, generates a mutant for that program.

Process

1. Each of these operators is applied to every statement of a program to which it is applicable, generating a number of mutants for the program.
2. Each of these mutants is then tested using the test suite
3. and noted the percentage of mutants killed by the test suite.

Mutation Score

$$mutation\ score = \frac{mutants\ killed}{total\ mutants}$$

The relational operator replacement rule takes an occurrence of a relational operator, <, =<, >, >=, =, or !=, replaces that occurrence with one of every other type of relational operator, and replaces the entire proposition in which that operator occurs with and .

Example

1. For statement if (x < y), seven mutants will be created, these operators lead to incorrect paths.
    a. if (x =< y)
    b. if (x > y)
    c. if (x >= y)
    d. if (x == y)
    e. if (x != y)
    f. if (true)
    g. if (false)
2. For statement x = 5

a. x = abs(5)
b. x = -abs(5)
c. x = failOnZero(5)

The Arithmetic Value Insertion rules takes an arithmetic expression and replaces it with its application to the absolute value function, the negation of the absolute value function, and fail-on-zero function, in which the fail-on-zero throws an exception if the expression evaluates to zero.

- This operator is designed to enforce the addition of test cases that consider the case in which every arithmetic expression evaluates to zero, a negative value, and a positive value.

**mutant operators for Java**

Arithmetic Operator Replacement
Replace each occurrence of an arithmetic operator +, -, *, /, **, and % with each of the other operators, and also replace this with the left operand and right operand.

- Example, replace x + y with x and with y.

Conditional Operator Replacement
Replace each occurrence of a logical operator &&, ||, &, |, and ^ with each of the other operators, and also replace this entire expression with the left operand and the right operand.

Shift Operator Replacement
Replace each occurrence of the shift operators <<, >>, and >>> with each of the other operators, and also replace the entire expression with the left operand.

Logical Operator Replacement
Replace each occurrence of a bitwise logical operator &, |, and ^ with each of the other operators, and also replace the entire expression with the left and right operands.

Assignment Operator Replacement
Replace each occurrence of the assignment operators +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, and >>>= with each of the other operators.

Unary Operator Insertion
Insert each unary operator +, -, !, and ~ before each expression of the correct type. For example, replace if (x = 5) with if (!(x = 5)).
Scalar Variable Replacement
Replace each reference to a variable in a program by every other variable of the same type that is in the same scope.

Bomb Statement Replacement
Replace every statement with a call to a special Bomb() method, which throws an exception.

- This is to enforce statement coverage, and in fact, only one call to Bomb() is required in every program block.

**Equivalent Mutants**
The Major Problem for Mutation analysis.
Given a program and a mutation of that program, the mutant is said to be an equivalent mutant if, for every input, the program and the mutant produce the same output.

- An equivalent mutant cannot be killed by any test case, because it is equivalent with the original program.
    - As an example of an equivalent mutant, consider the squeeze function.
- implies for many programs, an adequate test suite kills every mutant is unachievable.
    - In many practical applications of mutation analysis, a threshold score is targeted
        - For example, the tester aims to kill 95% percent of the mutants, and continues trying to kill mutants until that threshold is reached.
- Mutation testing is  the most successful test coverage criterion for finding faults, but mutation analysis is an expensive process.
    - Generating mutants manually is not feasible for anything other than small programs. It requires the tester to execute the test suite over every mutant, and most likely to iterate this execution a number of times.

Disadvantage

- Equivalent mutants are impossible to kill, and difficult to detect.
    - Once a test suite has been run over a collection of mutants, it is difficult to determine which mutants have not been killed due to the test suite being inadequate, or due to them being equivalent. Computing whether two programs are equivalent is undecidable,
    - So a manual analysis needs to be undertaken for many instances.

# Comparison

- Control Flow
    - Easy to see the relationships between the different control-flow techniques.
    - Decision/condition coverage subsumes both branch and condition coverage, as it is a combination of the two.
        - The subsumes relation is transitive, decision/condition coverage also subsumes statement coverage, because branch coverage does.
- Data Flow
    - Both data-flow criteria and coverage criteria select a set of paths that must be traversed by the test cases.

    - None of the data-flow coverage criteria subsume any of the coverage metrics related to conditions or decisions.
    - Easy to see the relationships between the different data-flow techniques.
- Multiple condition coverage and the criteria that it subsumes do not subsume any of the data-flow criteria, and do not subsume any of the data-flow criteria.
    - Multiple-condition coverage subsumes decision/condition coverage, and path coverage (if possible) subsumes branch coverage.
- The relationships between the different control-flow and data-flow criteria are less clear to see.
        - Condition coverage does not subsume statement coverage.

# Week 5

## Testing Modules

### Definition

1. State and Programs
    a. The data called the state is manipulated and accessed via a collection of operations on that state. Collectively, these are referred to as a module [1].
    b. A **module** does not require a state, but can be simply a collection of related operations.
    c. Using modules enforces a separation of interface and implementation, allowing programmers to think of these collections of operations as a black box.
    d. Specific details about the data are hidden from the user of the module, and a change in the underlying data structure has no impact on the program that uses it as long as the interface remains the same.
    e. **Operations are meaningless when separated** from their state and treated in isolation.
    f. Object-oriented programs are special classes of state-based programs, in that multiple instances of the data can be created.

### Testability

Testability: = Controllability + Observability

**Controllability**

- The controllability of a software artifact is the degree to which a tester can provide test inputs to the software.

**Observability**

- The observability of a software artifact is the degree to which a tester can observe the behavior of a software artifact, such as its outputs and its effect on its environment.

## Testability of State-Based Programs

The testability of a program is defined by the **observability** of the program, and the **controllability** of the program.

g. Example
   i. Initially, the stack is empty — it contains no elements. When created, the calling program determines a maximum size for the stack, passed as a parameter to the constructor.
   ii. An element can be pushed onto the stack if the size of the stack is not already equal to the maximum size, as determined when initialized. If full, pushing another element will result in the exception **StackFullException**.
   iii. The top element of the stack can be popped from the stack provided that the stack is not empty. Pushing an element and then popping the stack will result in the stack before pushing. If the stack is empty, popping will result in the exception **StackEmptyException**.
   iv. Finally, the stack can be checked to see if it is empty, and can be checked to see if it is full. These operations allow the calling program to avoid exceptions.

h. Example in Java
   i. The stack's maximum size is set by the program that uses it. Variables starting with an underscore (_) are private within the class and can't be directly accessed or seen.

This reduction in controllability and observability means that unit testing in isolation is not possible for the operations of the stack module.

- One way around these testability issues is to intrusively test the module.
  - The tester modifies the program code to give access to the data type that is hidden, therefore breaking the information hiding aspect of the module. **The program is tested, and the data type is hidden again.**
    i. it does not test the implementation that is to be used, but an altered version of it;
    ii. if the underlying data type in the module changes, then the tests must be changed as well;
    iii. it ignores the fact that operations acting on the same data are inherently linked, and must be tested together.
  - With modules such as the stack class, the **high-level of dependency between operations** implies that testing an operation in isolation would give us **little benefit**.
- Use a non-intrusive method

    - We use the operations defined by the module to set the state of the module to test the value that we want, perform the necessary test, and then use the operations again to query the result of the test.

Notes:
Many object-oriented languages provide us with constructs that are somewhat in between intrusive and non-intrusive testings.

- For example, in the Java programming language, variables inside classes can be declared as protected, which means that if we can inherit the class, then we can access the variables in the class. This allows us to passively observe the values of variables without changing the module itself.

## Unit Testing with Finite State Automata

The way that many modules, especially those that implement abstract data types, are abstracted is to envisage them as **finite state automata**.

- A **finite state automaton (FSA)** or finite state machine is a model of behavior consisting of a finite set of states with actions that move the automata from one state to another.
    - With regards to a module, each state in a FSA corresponds to a set of states in the module it is modeling.
    - The transitions between states correspond to the operations in the module.
    - Transitions can be prefixed with predicates specifying the preconditions that must hold for them to take place.
    - **FSAs are useful for modeling behavior of programs**, and can be used to derive test cases for such programs.
        - The states of the automaton derived from subsets of the states of the data encapsulated in the module, and the transition of the automaton are the operations of the module.
        - Test cases are selected to test sequences of transitions in this automaton.

### Construct a FSA

1. identify the states of the FSA
    a. Each state in the FSA corresponds to a set of states in the module.
        i. We want to consider states with the same effect on operations as being equivalent and collect them together into a single FSA state.
2. identify which operations are enabled in a state and which operations are not enabled in a state
    a. an operation is enabled if it can be called safely without error in a given state.
        i. For example, the pop and top operations are not enabled if the stack is empty, and the push operation is not enabled if the stack is full.
    b. From this information, we identify the start state(s), the exit state(s) (those that have no enabled operations).

3. identify the source and target states of every operation in the model
    a. To do this, we take every state and every operation that is enabled in that state, and calculate the state that results in applying that operation.

**Correlated Definition**

1. The **initial state** of this module is the undefined stack.
    a. Alternatively, one could remove the undefined state, and instead specify that the initial state is the empty state.
    b. The exit states are the exception states.
2. The **exit states** depend on the specification for the module.
    a. For example, popping an empty stack throws an exception, however, an alternative specification of the stack may assume that the stack is not empty when elements are popped, therefore, the behavior of popping an empty stack is undefined, and cannot be meaningfully tested. **In this case, there are no exit states** (there is always one operation that can be executed in any state).
3. The predicate in the square brackets on the transition labels represent the conditions that must hold for that transition to take place.
    a. These are not the preconditions of the operations themselves, but merely the conditions that must hold for that operation to change to the destination state.

## Deriving Test Cases from a FSA

The FSA of a module gives us the information that is required to set up a state for a test input of an operation that may require us to execute other operations in the module.

- If the states in the FSA represent the input states that we wish to test, then the transitions that move the state from the initial state to each state represent the operations that need to be executed.

The **aim** is to generate test sequences such that the paths in the automaton are exercised.

- A test case is no longer a single function call but may require a sequence of operation calls to force the module along a certain path.

**Traversal three criteria for a FSA**

1. **State coverage**
    a. Each state in the FSA must be reached by the traversal.
2. **Transition coverage**
    a. Each transition in the FSA must be traversed. This subsumes state coverage.
3. **Path coverage**
    a. Each path in the FSA must be traversed. This subsumes state coverage, but is impossible to achieve for a FSA with cyclic paths.

- State coverage is inadequate, because there are transitions in the FSA that will not be traversed, and therefore, operations that will not be tested for the test states that were derived.
- Transition coverage is feasible, straightforward to achieve, and it tests every test state that was derived, so this is generally the most practiced traversal technique.
- **Path coverage is impossible** for FSAs with cycles, so this is infeasible in many cases.

Using transition coverage, we repeatedly derive test sequences starting at the initial node until all transitions have been covered at least once.

- In some cases, this can be done using one long sequence, but in others, this is not possible; it is **more efficient to derive many short sequences than a few long sequences**.
    - For example, the stack FSA as two exit states, and one of these has two input transitions, which together imply that there must be at least three sequences.

## Intrusively Testing the State Transition Diagram

Intrusive testing will mean adding testing code to the module to measure and monitor the internal states of the module as it is tested.

- **The problem is it breaks encapsulation.**
- **Tests might break if the source code is modified.**

Testing code should be inserted into a module to check private state variables, monitor private function calls, and view intermediate stages within module operations.

- Sometimes in Java we would need to extend all elements of the hierarchy with testing code. In this case there is a great deal of additional code to write which may well further impact the properties of the entire system.
- In languages like C and C++ we can use the pre-processor to compile test code into the executable when required, or to omit the testing code when the program is to be released.
- The best cases are languages that provide semi-private access, such as the protected keyword in Java, or the friend keyword in C++. These give testers the ability to read and write to the variables without changing the program-under-test.

As a result of these problems, systematic intrusive testing is difficult to achieve, in fact, it is often impossible for some languages if we do not have access to the source.

## Non-intrusive Testing the State Transition Diagram

There will be occasions when we simply do not have access to the internal structure of a module or its hierarchy. In this case, or in the case where you simply need to test a module without adding code to a module, then we have non-intrusive testing.

- The idea is to test each of the transitions in the testing automaton implicitly by using other operations in the module to examine the results of a transition.
- The technique for testing involves deriving test sequences, using initialisers and transformers, to move the automaton to the required state.

- At each state, observe the value of the module using the observer operations. The only case where the observer operations are not used is in the exception states, because the execution of the module is considered to have terminated.
- Alternatively, and depending on the programming language, these exception states could be removed from the FSA, and considered as observer behavior.
- Many programming languages support the catching of exceptions, thereby allowing the execution to continue.
- **Usually a semi-intrusive approach is used.**
    - **Firstly design tests then based on test design logic**

Places that these can be tested on observer operations:

1. all observer operations could be used to observe the state after a call to any initialiser or transformer operation;
2. all observer operations could be used to observe the state any time a transition takes us to another state in the FSA;
3. all observer operations could be used to observe the state when an FSA state is being visited for the first time.

- The first of these seems like overkill, and would lead to a high number of test cases.
- For the second one, It's more practical to test observer operations only when a state in the Finite State Automaton (FSA) is encountered for the first time, given that each FSA state stands for multiple module states, each representing its own equivalence class.

### Problem for State Based Testing

1. It may take a lengthy sequence of operations to get an object into some desired state.
2. FSA-based testing may not be useful if the module is designed to accept any possible sequence of operation calls. This would result in a FSA with little structure, and require a prohibitively large number of test sequences to cover.
3. State control may be distributed over an entire application with operations from other modules referencing the state of the module under test. System-wide control makes it difficult to verify a module in isolation and requires that we identify module hierarchies that collaborate to achieve a particular functionality. Here the collaboration and behavior diagrams are the most useful.

## Testing Object-Oriented Programs

In this case, the modules are classes, and the operations are methods.

**Review on OOP**

1. Classes and objects which provide the main units for structuring.
    a. Classes are used as templates for creating objects.
    b. a class declaration introduces a new type while the objects are the elements of that type.

   c. Whenever a new object is created it is an instance, or element of that type.
   d. Every object in the class has
      - The methods defined by the class
      - The instance variables (or attributes) defined by the class
   e. **Classes and objects do not present any major obstacles in testing** on top of the encapsulation problems in testing modules.
2. Inheritance, which provides one of the key structuring mechanisms for object-oriented design and implementation.
   a. The parent is the more general class providing fewer methods or more general methods while the child specializes the parent.
   b. A child class inherits all of the properties of its parent, but the child class may override operations in the parent and extend the parent by adding more attributes and operations.
   c. In particular, generalization means that the objects of the child class can be used anywhere that the objects of the parent class can be used — but not the converse.
   d. Some languages, such as Java, support only single inheritance, in which a class may only inherit from at most one parent class. Others, such as C++, support multiple inheritance, in which a class may inherit from one or more parent classes. The parent class is called the superclass and the child is called the subclass.
   e. **Inheritance** is found only in object-oriented programming languages, but **does not pose any immediate problems in testing.**
3. Dynamic Binding or Polymorphism, which provides a way of choosing which object to use at run-time.
   a. The essence of **dynamic binding**, or as it is often called, **polymorphism in object-oriented languages**.
      - allows objects of different types to be treated as objects of a common super type, making it easier to create flexible and expandable software, especially in object-oriented programming.
   b. Dynamic binding and polymorphism are both found in some non-object-oriented languages, however, this is uncommon. The combination of inheritance and dynamic binding creates some real headaches for testers, or at least the combination of polymorphism and inheritance must be taken into account.

## Testing with Inheritance and Polymorphism

**In object-oriented testing** the view is that **classes are combined with object state and set of methods**.

- The internal states of the object become relevant to the testing.
  - The correctness of an object depends on the internal state of the object as well as the output returned by a method call.

- classes are the natural unit for testing object-oriented programs.
- Need to explore the effects of object-oriented testing in the presence of inheritance/generalization and polymorphism.

## Testing Inheritance Hierarchies

Inherited features **require re-testing**, because every time a class inherits from its parent the state and operations of the parent are placed into a new context — the context of the child.

- Multiple inheritance complicates this situation by increasing the number of contexts to test.

Example

```
class Parent{
    int number(){return 1;}
    float divide(int x){return x/number();}
}
class Child extends Parent{
    int number(){ return 0;}
}
```

1. Parent is at the top of an inheritance hierarchy, so we derive test cases for this class first, and test it.
2. When testing the Child class, we need to retest the number() method because this has been modified directly. However, we also need to retest the divide() method because it directly references number(), which has changed.

### Building and Testing Inheritance Hierarchies

When dealing with inheritance hierarchies it is important to consider both testing and building at the same time. Reason:

1. Keep control of the number of test cases and test harnesses that need to be written.
2. Make sure that we know where faults occur within the inheritance hierarchy as much as possible.

A first approach to inheritance testing involves **flattening the inheritance hierarchy.**

- Each subclass is tested as if all inherited features were newly defined in the class under test
    - so we make absolutely no assumptions about the methods and attributes inherited from parent classes.
- Test cases for parent classes may be reused after analysis but many test cases will be redundant and many test cases will need to be newly defined.

A second approach to testing and building is incremental inheritance-based testing.
**Incremental building** and testing proceeds:

1. test each base class by
    a. Testing each method using a suitable test case selection technique
    b. Testing interactions among methods by using the techniques.
2. consider all sub-classes that inherit or use (via composition or association) only those classes that have already been tested.
    a. A child inherits the parent's test suite which is used as a basis for test planning. We only develop new test cases for those entities that are directly, or indirectly, changed.

Advantage
Incremental inheritance-based testing reduces the size of test suites, but there is an overhead in the analysis of what tests need to be changed.

- It reduces the number of test cases that need to be selected over a flattened hierarchy.
    - If the test suite is structured correctly, test cases can be reused via inheritance as well, which provides the same benefits for conceptualisation and maintenance for test suite implementations as it does for product implementations.

Inheritance-based testing can also be considered to be a form of regression testing where the aim is to minimize the number of test cases needed to test a class modified by inheritance.

**Implications of Polymorphism**
In procedural programming, procedure calls are statically bound — we know exactly what function will be called at compile time — and further, the implementation of functions does not change (well, not unless there is some particularly perverse programming) at runtime.

- In the case of object-oriented programming, each possible binding of a polymorphic class requires a separate set of test cases. The problem for testing is to find all such bindings – after-all the exact binding used in a particular object instance will only be known at run-time.

Dynamic binding also complicates integration planning. Many service and library classes may need to be built and tested before they can be integrated to test a client class.

There are a number of possible approaches to handling the explosion of possible bindings for a variable.

- One approach is to try and determine the number of possible bindings through a combination of static and dynamic program analysis.
    - However, this approach is not foolproof and is biased heavily towards the data used to generate the bindings.

# Week 6

## Property-based Testing and Test Oracles

A test oracle is someone or something that determines whether or not the program has passed or failed the test case. Or it can be another program that returns a “yes” if the actual results are not failures and “no” if they are.
A test oracle is:

1. a program;
2. a process;
3. a body of data;

that determines if actual output from a program has failed or not.

- Ideally an oracle should be **automated** (no human intervention at runtime) because then we can execute a larger volume of test cases and gain greater coverage of the program, but this is often non-trivial in practice.
    - What we can do is to define properties that should hold in our program for specific inputs.

### Active and Passive Test Oracles

An automated oracle can be placed into one of two categories:

1. Active oracle
    a. A program that, given an input for a program-under-test, can generate the expected output of that input.
2. Passive oracle
    a. A program that, given an input for a program-under-test, and the actual output produced by that program-under-test, verifies whether the actual output is correct.

2 Reasons for Passive oracles are generally preferred

1. Passive oracles are typically easier to implement than active oracles.
    a. Passive one not only saves the tester some time, but also means that there is less chance of introducing a fault into the oracle itself.
2. Passive oracles can handle non-determinism.
    a. Using an active oracle, which predicts outcomes, often yields results different from the actual output because it anticipates specific behaviors. In contrast, a passive oracle, checking if the output is right without predicting it, doesn't face issues from unpredictability (non-determinism).

### Types of Test Oracle

1. Formal, executable specifications

   a. Formal specifications, using precise mathematical language, are more suitable for developing testing oracles than informal, natural language descriptions. They can serve as active oracles, forecasting results through simulation, or passive ones, confirming that real outputs meet specified expectations. However, these methods are typically **used only in critical systems** where safety, security, or mission is a high priority due to their complexity and rigor.

2. **Solved Examples**
   a. Often sourced from texts or references, or developed by hand, they are valuable for testing, especially in complex areas where generating expected results is as difficult as the program itself, and manual calculation needs specific expertise that testers might not have.
   b. Using recorded data like tables, documents, or graphs as testing oracles is effective. You can compare the test input and actual output with this data to check correctness.
   c. However, these oracles limit the inputs to available examples. Still, they're **widely used** because such data is plentiful in many domains.
3. **Metamorphic Oracles**
   a. Sometimes, we can use "metamorphic" properties to cross-check tests. 'Metamorphic' means if two different inputs are related by a certain property (p), their outputs after running through the program will be related by another property (q). The connection between p and q is called the metamorphic relation.
   b. Use known relationships between different inputs and outputs to validate a program's behavior. Essentially, if you know how changing an input should affect the output, you don't need to know the exact output to confirm the program is working correctly.
   c. Metamorphic properties are common in programs, and metamorphic oracles aren't just used for testing numerical software. They're also **applied in diverse fields** like bioinformatics, search engines, machine learning, medical imaging, and web services.
   d. Metamorphic Testing requires good domain knowledge of a problem.
   e. Different metamorphic relations can have different fault detection capability.
   f. Metamorphic relations can be combined.
4. Alternate implementations
   a. An alternate implementation is when you create a different version of a program to compare expected outputs. However, it's **not perfect because different versions often share the same errors**. So, using an alternative might miss some faults present in the original program.
   b. A useful method is creating a simplified, partial version of the program. This version doesn't have all the features of the full program and doesn't focus on speed or memory use, but it helps in identifying issues in the main program.

   c. Such an approach restricts the test inputs that can be used, but is often sufficient to find many faults in a system.

5. **Heuristic oracles**
   a. These are oracles that provide approximate results for inputs, and tests that "fail" must be scrutinized closely to check whether they are a true or false positive.
   b. Heuristic oracles involve identifying and using patterns between inputs and outputs in complex programs to facilitate testing.
   c. **In reality, most oracles are heuristic**, in that none of them really replicates the expected behavior of the corresponding program.
      i. However, we use the term heuristic oracle to refer to oracles that are designed based on some heuristics that are not complete, maybe not sound, but test some properties of software under test.
6. **The Golden Program**
   a. The ultimate source for a testing oracle but **rare in practice**. The golden program is an executable specification, a previous version of the same program, or tested in parallel with a trusted system that delivers the same functionality.
   b. Still, the golden program is not a pipedream. In industry, it is common to use the previous release of a piece of software as a test oracle.
7. **Oracle derivation/Human Oracle**
   a. Usually, human testers generate test cases and establish oracles based on the product's design and specifications. Often, testers have to estimate the expected software behavior themselves.
   b. Advanced testing methods like model-based testing automate the creation of both inputs and oracles. Instead of manually creating test cases, testers build a formal, executable model outlining the program's expected behavior. This model automatically produces test inputs and calculates expected outputs through simulation, blending elements of basic and complex oracles. While the model serves as an alternate, abstract version of the software, its higher abstraction level decreases the chance of faults occurring on the same inputs.
   c. Model-based testing is no more expensive than manual test case generation, and in many cases, is significantly more efficient, and is as successful for locating faults.

# Property-based testing

Property-based testing combines the idea of (usually passive) test oracles with techniques such as random testing to run a large amount of test inputs, and check their output against certain properties.

- Not techniques like equivalence partitioning
  - It is time consuming to find good tests;

    - Aim to select only a small number. This is not so good for testing the robustness of software.

### Defining property-based templates

1. generating random inputs
2. such that a condition holds
3. checking whether a property holds on those inputs.

### Useful Tools

A better way to implement property-based testing, rather than doing it by hand, is to use tools designed specifically for the task.

1. Hypothesis is a property-based testing framework for Python.
2. jqwik is a property-based testing framework for Java.

Both frameworks help to generate random data, filter out inputs according to the condition, and to specify properties.

- Tools like jqwik also do more than random testing.
    - For example, if they find an example that does not hold, they try to search for a simpler, smaller example that still violates the property. Because this is smaller and simpler, it can make debugging easier.

### Advantages of Property-Based Testing

1. It can generate a large volume of tests with minimal human oversight and cost.
2. It encourages us to make our assumptions explicit by defining properties.
3. It can find things that a human tester does not think of.

### Disadvantages of Property-Based Testing

1. The coverage of the input domain is not systematic, so it is unlikely to test edges cases like boundary values unless explicitly told to, and is unlikely to achieve good coverage of the software unless specifically guided.
2. It only tests the properties that it is given, which are (usually) partial properties. While specifying test oracles that are complete is possible, it is uncommon.

Based on these features, property-based testing and example-based testing complement each other, so they should be used together; rather than relying on just one technique.

# Testing-and-Integration

The order in which modules are integrated and tested can make the process of finding faults easier if a good strategy is chosen and may result in fewer latent faults [1].

The process of developing and debugging your programs, building your programs, validating and verifying your programs and testing your programs are interrelated and depend upon each other.

## Integrating the System

In the process of designing a system, you will have decomposed your system into modules, packages or classes.

- Integrating the system refers to putting the modules, packages or classes together to create subsystems, and eventually the system itself.
- Integration and testing go together.
    - Good integration strategies can help you to minimize testing effort required. Poor integration strategies generally cause more work, and lead to lower quality software.
- Normally, a system integration is organized into a series of builds.
    - Each build combines and tests one subset of the modules, packages or objects of the system.
- The aim for most integration strategies is to divide the modules into subsets based on their dependencies.
    - Typical build strategies then try to integrate all those modules with no dependencies first, the modules that depend on these second, and so on.

### The Big-bang Integration Strategy

The Big-Bang method involves coding all of the modules in a sub-system, or indeed the whole system, separately and then combining them all at once.

- Each module is typically unit tested first, before integrating it into the system.
- The modules can be implemented in any order so programmers can work in parallel.
- Once integrated, the completely integrated system is tested.

Issues

1. If the integrated systems fails a test then it is often difficult to determine which module or interface caused the failure.
    a. There is also a huge load on resources when modules are combined (machine demand and personnel).
2. The number of input combinations becomes extremely large
    a. testing all of the possible input combinations is often impossible and consequently we there may be latent faults in the integrated sub-system

### A Top-Down Integration Strategy

The top-down method involves implementing and integrating the modules on which no other modules depend, first.

The modules on which these top-level modules depend are implemented and integrated next and so on until the whole system is complete.

- requires stubs to be written
    - A stub is an implementation used to stand in for some other program.
    - A stub may simulate the behavior of an existing implementation, or be a temporary substitute for a yet-to-be-developed implementation.

Advantages

1. over a Big-Bang approach
    a. the machine demand is spread throughout the integration phase.
2. If a module fails a test then it is easier to isolate the faults leading to those failures.
    a. Because testing is done incrementally, it is more straightforward to explore the input for program faults.

## The Bottom-Up Integration Strategy

The bottom-up method is essentially the opposite of the top-down.
Lowest-level modules are implemented first, then modules at the next level up are implemented forming subsystems and so on until the whole system is complete and integrated.

- A common method for integration of object-oriented programs, starting with the testing and integration of base classes, and then integrating the classes that depend on the base classes and so on.
- Requires drivers to be written.
    - A driver is a piece of code used to supply input data to another piece of code.
    - The piece modules being tested need to have input data supplied to them via their interfaces.
    - The driver program typically calls the modules under test supplying the input data for the tests as it does so.

## A Threads-Based or Iterative Integration Strategy

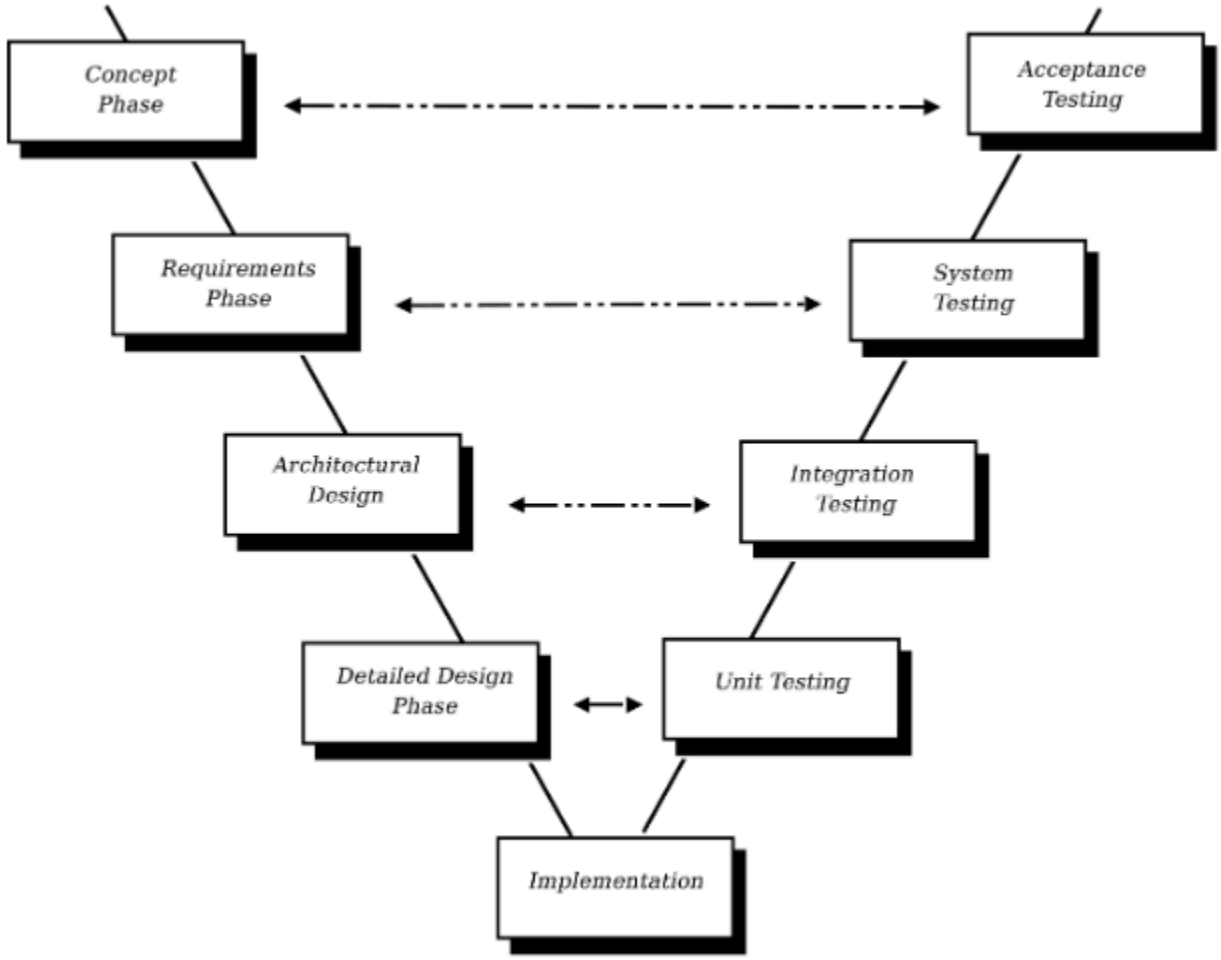
The threads-based integration method attempts to gain all of the advantages of the top-down and bottom-up methods while avoiding their weaknesses.
The idea is to select a minimal set of modules that perform a program function or program capability, called a thread, and then integrate and test this set of modules using either the top-down or bottom-up strategy.

- Ideally, a thread should include some I/O and some processing where the modules are selected from all levels of the module hierarchy.
    - Once a thread is tested and built other modules can be added to start building up a complete system.
- Common in agile methodologies.
    - Threads are focussed on user requirements, or user stories.
        - Given a coherent piece of functionality from a set of user requirements, we build that small piece of functionality to deliver some value to the user.
- This is **NOT an integration strategy on its own**

- we will still have to bring together parts of this as we go, using top-down and bottom-up strategies.

## Types and Levels of Testing

There are various types of testing that systems typically undergo. Each type of testing is aimed at detecting different kinds of failures and making different kinds of measurements.




### Unit Testing(UT)

The first task in unit testing is to work out exactly what a unit is for the purpose of testing.
In most cases, a unit is a module

- a collection of procedures or functions that manipulate a shared state.
    - In these cases, the module is tested as a whole, because changes in one procedure can affect others.
- In other cases, a unit is a single procedure or function.
    - In these cases, the only inputs and outputs are parameters and return values — there is no state.

Units, if they are functions or classes, are tested for correctness, completeness and consistency.

- Unit testing measures the attributes of units.
    - They can be tested for performance but almost never for reliability;
    - units are far too small in size to have the statistical properties required for reliability modeling.
- Developers do the test
    - Using White-Box Testing Method
    - UT Frameworks, drivers, stubs, and mock/fake objects are used

**If each unit is thoroughly tested before integration** there will be **far fewer faults** to track down later when they are harder to find.

## Integration Testing(IT)

Integration testing tests collections of modules working together.

- Which collection of modules are integrated and tested depends on the integration strategy.

The aim is to test the interfaces between modules to confirm that the modules interface together properly.

- Integration testing also aims to test that subsystems meet their requirements.
- Integration tests are derived from high level component designs and requirements specifications.
- Either developers or independent testers
    - Using any of black box, white box and gray box testing method
    - Test drivers and test stubs are used to assist in integration testing.

## System Testing(ST)

System testing tests the entire system against the **requirements, use cases or scenarios** from requirements specification and design goals.

1. System Functional Test
    a. tests the entire system against the functional requirements and other external sources that determine requirements such as the user manual.
2. System Performance Test
    a. test the non-functional requirements of the system.
        i. For example, the load that the system can handle, response times and can test usability (although this latter testing is more like a survey than what we would call an actual test).
3. User Acceptance Test
    a. is a set of tests that the software must pass before it is accepted by the clients.
        i. This is typically a form of validation, whereas testing against the specification is a form of verification.

Normally independent testers perform ST, using black box testing method.

## Acceptance Testing

The level of the software testing process where a system is tested for acceptability.
The purpose of AT is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

- Product management, sales, customer support and customer do the tests
    - Usually use black box testing, the testing is done ad-hoc and non-scripted.

## Regression Testing

Programs undergo changes but the changes do not always effect the entire program. Rather certain functions or modules are changed to reflect new requirements, system evolution or even just bug fixes.

- After making a change to a system, testing only the part of the system that has changed is not enough.
    - A change in one part of a system can effect other parts of the system in ways that are difficult to predict.

Therefore, after any change, the entire test suite of a system must be run again. This is called regression testing.

- **Regression testing is one of the key reasons** for wanting to be able to execute test suites **automatically**.
    - To **test an entire system by hand is costly** even for the smallest of systems, and infeasible for many others.

## Exploratory Testing

Exploratory testing is a form of unstructured testing.

- Experienced testers with knowledge of the application use their software engineering background and learnings from the testing process to explore the software and identify issues.

It aims to exploit human insight to find faults. By exploring the application, the human tester gains insight that they could not gain by just writing tests.

- It is typically conducted at a system level (but not always)
    - having people sit and explore the application, trying out different parts, observing what happens, and then repeating.
- Charter
    - Defines mission
- Session-based test management
    - Defects + Notes + interviews of the testers

During exploratory testing, do:

1. Mission of testing should be very clear
2. Keep notes on what needs to be tested, why it needs to be tested and the assessment of the product quality.
3. Tracking of questions and issues raised during exploratory testing.
4. Better to pair up the testers for effective testing.
5. The more we test, more likely to execute right test cases for the required scenarios

Very important to document and monitor the following

1. Test Coverage
    a. Whether we have taken notes on the coverage of test cases and improve the quality of the software
2. Risks
    a. Which risks needs to be covered and which are all important ones?
3. Test Execution Log
    a. Recordings on the test execution.
4. Issues / Queries
    a. Take notes on the question and issues on the system.

Advantages

1. During testing, testers gain insights that can help them understand the requirements more deeply than structured testing allows, leading to new, unconsidered findings.
2. Structured tests require both a test input and an expected output.
    a. So, the tester will be focused on the test output, and may completely ignore other things that are going on.
        i. They are less likely to notice other anomalies on the page compared to an exploratory tester.
3. **Less structured in nature**

Disadvantages

1. It does not really offer any type of structure around the testing
    a. such as how to maximize chances of finding faults.
2. It is **difficult to repeat**, and difficult to run regression tests
    a. because we do not really record what is happening.
3. **No guarantee all faults/defects will be detected.**

Exploratory testing is best used in combination with structured techniques. Although it is less scientific in the way test selection and execution is done, **its advantage is the use of human insight to find faults**.

# Week 7

## Security Testing

### Penetration Testing

The aim of penetration testing is to find vulnerabilities.
A vulnerability is a security hole(**fault**) in the hardware or software (including operating system) of a system that provides the possibility to attack that system;

1. e.g., gaining unauthorized access.
2. Vulnerabilities can be weak passwords that are easy to guess
3. buffer overflows that permit access to memory outside of the running process
4. unescaped SQL commands that provide unauthorized access to data in a database.

### SQL Injection

```
SELECT * FROM really_personal_details WHERE email = '$EMAIL_ADDRESS'
```

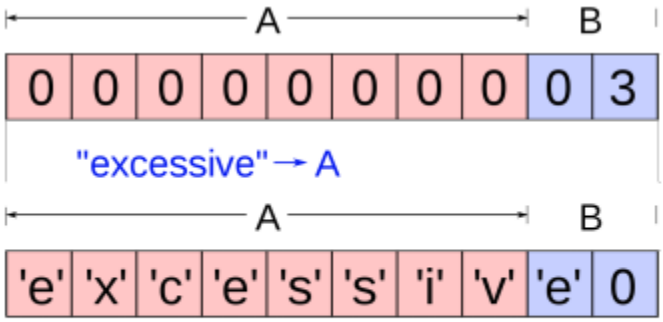
Leading to

```
SELECT * FROM really_personal_details WHERE email =
'myemail@somedomain.com' OR '1=1'
```

### Buffer overflows

These occur when data is written into a buffer (in memory) that is too small to handle the size of the data.

- occurs when a **large piece of data** is written into a **small buffer** (in memory).
    - The data overruns the buffer's boundary and overwrites adjacent memory locations.
    - If carefully planned, attacker-generated data and code can be written into the memory and be executed.




Data is written into A, but is too large to fit within A, so it *overflows* into B [Wikipedia].

- In some languages, such as C and C++, the additional data simply overwrites the memory that is located immediately after the buffer.
    - If carefully planned, attacker-generated data and code can be written here.

In an attack, the string would contain a payload that could be a malicious program, which would have privileges equivalent to the program being executed; e.g., it could have superuser/admin access.

### Buffer overflow write and Buffer overflow read

Reading is more critical than writing, because it happens silently.

**Mitigation**

1. defensive programming that checks for array bounds
2. using programming languages that do this automatically
3. finding and eliminating these problems.
4. We can find these problems using reviews and inspections, letting hackers find them for us (not recommended)
5. using verification techniques(eg. Fuzzing testing)

## Fuzzing

Fuzz testing (or fuzzing) is a (semi-)automated approach for penetration testing that involves the randomisation of input data to locate vulnerabilities.

- At a high level, fuzzing refers to a process of **repeatedly** running a program with **automatically generated inputs** that may be syntactically or semantically malformed.
- Typically, a fuzz testing tool (or fuzzer) generates many test inputs and monitors the program behavior on these inputs, looking for things such as exceptions, segmentation faults, and memory leaks; rather than testing for functional correctness.
- Typically, this is done live
    - One input is generated, executed, and monitored, then the next input, and so on.

A fuzzer can be categorized in three ways:

1. **Generation-based vs mutation-based**
    a. inputs are generated from scratch with grammars/input models or by modifying existing inputs.
2. A fuzzer can be dumb (**unstructured**) or smart (**structured**) depending on whether it is aware of input structure.
3. A fuzzer can be **white-, gray-, or black-box**, depending on the program structure information it is aware of.

### Three Techniques of Fuzzing

1. Random testing
    a. tests generated randomly from a specified distribution.
2. Mutation-based fuzzing

   a. starting with a well-formed input and randomly modifying (mutating) parts of that input
3. Generation-based fuzzing
   a. using some specification of the input data, such as a grammar of the input.

## Random Testing

In random fuzzing, tests are chosen according to some probability distribution (possible uniform) to permit a large amount of inputs to be generated in a fast and unbiased way.

Advantages

1. It is often (but not always) cheap and easy to generate random tests, and cheap to run many such tests automatically. **Generally Fast**.
2. It is **unbiased**, unlike tests selected by humans.
   a. This is useful for pentesting because the cases that are missed during programming are often due to lack of human understanding, and random testing may search these out.

Disadvantages

1. A prohibitively **large number of test inputs may need** to be generated in order to be confident that the input domain has been adequately covered.
2. The distribution of random inputs simply misses the program faults
   **a. Might miss many vulnerabilities**
3. It is **highly unlikely to achieve good coverage**.
   a. If we take any non-trivial program and blast it with millions of random tests, it is unlikely that we will achieve good coverage, where coverage could be based on any reasonable criteria (control-flow, mutation, etc.).

In some cases, a random testing tool will likely spend most of its time just testing the false case over and over, which is not particularly useful.

- A standard way to address this is
   a. to (if possible) measure the code coverage achieved by your tests after a certain amount of time, and look for cases like these.
   b. Then, modify your random testing tool to first send some data with a correct checksum,
   c. then use random testing for the remainder of inputs.
   d. Rinse and repeat.

## Mutation-based Fuzzing

Mutation-based fuzzing is a simple process that takes valid test inputs, and mutates small parts of the input, generating (possibly invalid) test inputs.

- The mutation (not to be confused with mutation analysis discussed in Section Mutation Analysis) can be either random or based on some heuristics.

Example

1. 'myemail@somedomain.com' could be mutated to things such as:
    a. 'myemail@somedomain.com' OR '1=1'
    b. 'myemail@somedomain.com' AND email IS NULL
    c. 'myemail@somedomain.com' AND username IS NULL
    d. 'myemail@somedomain.com' AND userID IS NULL
    e. These last three attempt to guess the name of the field for the email address. If any of these give a valid response, we know that we guess the name of the field correctly; otherwise a server error will be thrown.

Advantages

1. It generally achieves higher code coverage than random testing.
    a. While issues such as the checksum issue discussed earlier still occur, they often occur less of the time if the valid inputs that are mutated have the correct values to get passed these tricky branches.
        i. Even though the mutated tests may change these, some will change different parts of the input, and for example, checksum will still be valid.

Disadvantages

1. The success is highly dependent on the valid inputs that are mutated.
    a. Good quality inputs (aka seeds) to mutate
    b. Good heuristics to mutate these inputs
2. It still suffers from low code coverage due to un likely cases (but not to the extent of random testing).
3. Challenges on input **validity & integrity**
    a. Integrity checks (e.g., checksums)
    b. Semantic relationships (e.g., length-of, size-of)
    c. Syntactic rules

**Tools to support mutation-based fuzzing**

1. Radamsa (akihe/radamsa)
    a. used to test how well a program can withstand malformed and potentially malicious inputs.
    b. It works by reading sample files of valid data and generating interestingly different outputs from them.
2. zzuf (http://caca.zoy.org/wiki/zzuf)
    a. a commonly-used tool for “corrupting” valid input data to produce new anomaly tests.
    b. It uses randomisation, and has controllable properties such as how much of the input should be changed for each test.
3. Peach (http://www.peachfuzzer.com/)
    a. A well-known fuzzer that supports mutation fuzzing, and has reached a level of maturity that makes it applicable to many projects.

# Week 8

## Generation-based Fuzzing

Generation-based fuzzers (or intelligent fuzzers) typically generate their own input from such existing models, rather than mutating existing input (although many tools combine the two approaches).

**The strategies applied**, such as which parts to change and how fine-grained the changes are, **affect the "intelligence" of the fuzzer.**

- Typically, generation-based fuzzers have some information about the format of the input that is required;
    - for example, a grammar of the input language (e.g., SQL grammar), knowledge of the file format.
- Using this knowledge, a generated-based fuzzer can create inputs that preserve the structure of the input, but it can randomly or heuristically modify parts of the input based on that knowledge.
    - Therefore, instead of randomly modifying parts of a string representing an SQL query, it can **produce syntactically-correct SQL**, but **with random data** within that structure.
- By knowing the input protocol and the interactions that are required, a generation-based fuzzer can more intelligently select inputs.
    - For example, by knowing a protocol for a web server, it can behave as a true web client would, allowing generation of correct, dynamic responses to server responses, rather than just random data.

A generation-based fuzzer would know the HTTP and SOAP protocols, so would first generate the skeleton for a parsable query, and then generate the values in between the tags of the SOAP query (randomly or heuristically).

- For example, it may randomly change the value in the firstName tag to a much longer string.
    - This would result in a SOAP query that is longer than specified in the Content-Length HTTP parameter (372 chars), perhaps resulting in a case where the SOAP query overflows the buffer that is allocated
        - presumably 373 chars (although one would hope that the server is using a library that does not do this!).

Advantages

1. Knowledge of the input protocol means that valid sequences of inputs can be generated that explore parts of the program, thus generally **giving higher coverage**.

Disadvantages

1. Compared to random testing and mutation fuzzing, it **requires some knowledge** about the input protocol.
2. The **setup time is generally much higher**, due to the requirement of knowing the input protocol
    a. although in some cases, the grammar may already be known (e.g., XML, RFC).

**Tools to support generation-based Fuzzing**

1. Peach(http://www.peachfuzzer.com/)
    a. mentioned also at a mutation fuzzer, supports both mutation and generation-based fuzzing.
    b. For smarter generation-based fuzzing, it also monitors feedback sent to it via network protocols (for fuzzing e.g., web servers) to intelligent select new tests.
2. Generation-based black-box fuzzer
    a. Domato
        i. fuzzing Web browsers (e.g., Safari, Chrome)
        ii. https://github.com/googleprojectzero/domato
    b. FeroxFuzz
        i. a structure-aware HTTP fuzzing library
        ii. https://github.com/epi052/feroxfuzz
    c. BooFuzz
        i. fuzzing file processing programs and network protocols
        ii. https://github.com/jtpereyda/boofuzz

# Memory Debuggers

A memory debugger is a tool for finding memory leaks and buffer overflows.
**Memory debuggers are important in fuzzing**, especially in languages with little support for memory management.

- because anomalies such as buffer overflows are difficult to observe using system behavior.
    - If the overflow is just by a few characters, it would be difficult to detect unless that particular part of memory is accessed again, which may not be the case.

## 4 Issues memory debugger monitoring

1. Uninitialised memory
    a. references made to memory blocks that are uninitialised at the time of reference.
2. Freed memory
    a. reads and writes to/from memory blocks that have been freed.
3. Memory overflows
    a. writes to memory blocks past the end of the block being written to.
4. Memory leaks

a. memory that is allocated but no longer able to be referenced.

Memory debuggers typically work by modifying the source code at compile type to include specific code that checks for these issues

- for example, by keeping track of a buffer size, and then inserting code directly before a write to check whether the memory being written is larger than the target buffer.

**Performance Cost**

While monitoring for these properties is useful, it is generally at a **high performance cost**. The overhead of keeping track of the allocated memory, plus the checks made, is expensive.

- For example, programs running using the well-known Valgrind tool run between 20-30 times slower than with no memory debugging.

**Tools to support memory debugger**

1. Valgrind(and its tool memcheck)
    a. A well-known open-source memory debugger for Linux, Mac OS, and Android.
        i. It works by inserting monitoring code directly into the source, and replaces the standard C memory allocation tools (e.g., alloc, malloc) with its own implementation that monitors references and memory block sizes, etc.
2. Rational Purify
    a. A commercial memory debugger for Linux, Solaris, and Windows.

## Undefined Behavior

Buffer overflows are a form of undefined behavior

- something that a program does that causes its future behavior to be unknown.
- It might continue working or it might do something totally unpredictable
    - Such as executing attacker-supplied code in the case of a successful remote code execution attack.

Programming languages like C define many forms of undefined behavior, besides buffer overflows. Examples of undefined behavior in C include:

1. Dividing by 0
2. Dereferencing a NULL pointer
3. Overflowing a signed integer
4. Underflowing a signed integer
5. plus many more.

Testing for undefined behavior, as with buffer overflows, requires special tool support.

- Because the behavior a program exhibits when performing an operation involving undefined behavior cannot be relied upon.

- Compilers use this fact when optimizing programs which can sometimes lead to very surprising results.

Whenever this bit of code doesn't perform undefined behavior (i.e. doesn't cause signed overflow), it is equivalent to doing nothing.

- Whenever it does perform undefined behavior, the compiler is allowed to have the program do whatever it wants.
- Thus removing this section of code is entirely legal: in all cases where undefined behavior doesn't occur the program behaves identically.

In general, a compiler is allowed to perform any transformation on a program so long as it ensures that the program's behavior remains unchanged in all cases where it doesn't exhibit undefined behavior.

- Bugs similar to the one in the program above have led to vulnerabilities in Linux kernel drivers when the compiler has optimized away an overflow test of the above form that was being used to guard against accessing a buffer out-of-bounds.

The unpredictability of how a program will behave when performing undefined behavior means that undefined behavior is important to test for.

- However, for the same reason, testing for undefined behavior requires special help from the compiler.
- Essentially, the compiler produces a program that includes checks to ensure that execution is halted with an error whenever undefined behavior is about to occur.
- As with memory debuggers, these checks necessarily incur some **performance overhead**.

**Tools to support Code Compiling**

1. Clang C compiler
    a. provides options for compiling code to enable it to be tested for undefined behavior.
        i. With these options turned on, Clang builds programs so that they generate an error whenever they perform undefined behavior.
    b. Example

```
$ clang signed_overflow.c -o signed_overflow \
        -fsanitize=undefined-trap -fsanitize-undefined-trap-on-error
$ ./signed_overflow 2147483647 10
Illegal instruction: 4
$
```

Here we see that the Illegal instruction error is generated when signed overflow occurs.

# Code Coverage-Guided Fuzzing

The three fuzzing techniques discussed in the previous chapter **(random, mutation and generation-based black-box fuzzing)** **all suffer from the same drawback**

- The strategy that they use to generate inputs for the program under test has no feedback loop.
    - Each of these methods generates inputs for the program under test but it does so in a way that is generally blind to (or unaware of ) what the program is doing.
- One solution for limitations of black-box fuzzing techniques
    - by allowing the fuzzer to monitor the execution of the program under test and to use this information to guide its decisions about what inputs to generate next.

## Definition

Monitor the code coverage achieved by each input, also called a greybox fuzzing technique.

- because it operates with partial knowledge of the program-under-test
    - specifically by being able to observe what parts of the program are covered (i.e. executed) by each input.

### American Fuzzy Lop (AFL)

Working Flow

1. begins with some seed inputs (or by generating an initial random input).
2. It maintains a list of interesting inputs and generates new inputs for the program-under-test by selecting an input from this list and mutating it.
3. When running the program on this new input, it monitors the program to determine what execution path the input followed.
    a. If a new part of the program was executed that hasn't yet been executed by previous inputs
        i. The new input is considered interesting and is added to the list of interesting inputs.
    b. Otherwise it is discarded
        i. because it did not allow the fuzzer to learn any new information about the program-under test.

Algorithm

1. $interesting \leftarrow initial\_test\_seeds$
2. $seen \leftarrow \emptyset$
3. while true do
4. Choose an input i from interesting
5. $input \leftarrow MUTATE(i)$
6. $path \leftarrow EXECUTE(input)$
7. if $\{path\} \not\subset seen$ then
8. $interesting \leftarrow interesting \cup \{input\}$
9. $seen \leftarrow seen \cup \{path\}$

- In reality, various heuristics are used to choose which input i should be mutated at each iteration and the specific mutation strategy to apply to i.
- Also, rather than tracking the precise path that the execution follows, for performance reasons, most coverage-guided fuzzers track more coarse-grained information
    - (such as which branches in the program were taken, with an approximate count of how many times each was taken, and so on).

A massive advantage compared to other fuzzing methods

- This argument can be repeated to consider all of the ways in which the ultimate input "bad!" might be progressively discovered by a coverage-guided fuzzer.
    - Within some thousands of mutations, we can expect even a naive coverage guided fuzzer to uncover the fault in this program.
- This is also the reason coverage guided fuzzing is popular.
    - Coverage-guided greybox fuzzers like AFL and lib-Fuzzer have been used to find many security vulnerabilities across a range of programs.
        - They are especially good at reaching deep program paths that involve incremental equality tests like the good bad program above.

**The Problem of Multi-Byte Equality Tests**

In some case we can expect a coverage-guided fuzzer to perform no better than a random fuzzer since its feedback mechanism offers it no additional information

- A coverage-guided fuzzer is not given any feedback until it happens to discover the input "bad!", which is incredibly unlikely.
- all inputs given to this program follow the same path except the single input "bad!".

As a result, coverage-guided fuzzers can struggle to generate inputs that cause checksum tests and other multi-byte equality tests to succeed.

Solutions:

1. rewrite such checks to be performed incrementally (one-byte-at-a-time)
2. remove the check altogether.

Both techniques have been used to aid coverage-guided fuzzing to find real security vulnerabilities.

**Performance Cost**

In order to monitor the program-under-test to determine which path an execution follows, one of two strategies is typically used. Both of these choices impose a performance overhead.

1. If source code for the program-under-test is available
    a. compile it with a special compiler that adds extra instrumentation code to the compiled object (binary) code.
        i. This instrumentation tracks the execution path that is followed by the program and reports it back to the fuzzer.
2. If source code is unavailable

    a. the program must be run in a virtual machine or an emulator (i.e. it has to be interpreted, rather than executed natively), to allow its execution to be tracked.

The **first** is **generally faster** and so is preferable when source code is available.
The **second** has the **advantage of being applicable to privileged code** like operating system kernels which are otherwise difficult to fuzz.

Advantages

1. Good coverage can be achieved even with little to no knowledge of the input format
2. Low set-up time
3. The technique can be widely applied

Disadvantages

1. **Coverage can still be limited** in programs that perform multi-byte equality tests (e.g. that compare two 4-byte integers)
    a. since such comparisons provide no incremental feedback to the fuzzer to allow it to discover the needed input to pass the test.
2. **Performance overhead is higher** than with fuzzing techniques that use no feedback loop.
3. **For** programs whose **input format is known**, **generation-fuzzing** and similar techniques are likely to **perform better**
    a. Because they can avoid the drawbacks of coverage-guided fuzzing.

**Limitation of code coverage-guided greybox fuzzer**

1. Rely on *random* mutation operators to generate new inputs
2. Can't prove the absence of faults/bugs

**Tools to support Code Coverage-Guided Fuzzing**

1. American Fuzzy Lop (AFL)
    a. https://github.com/google/AFL/
    b. popularized coverage-guided fuzzing.
2. libFuzzer
    a. https://llvm.org/docs/LibFuzzer.html
    b. A coverage-guided fuzzer built into the LLVM C compiler, Clang.

# Week 9

## Symbolic Execution

Symbolic execution is one approach to software verification that aims to do just that.

- The idea behind symbolic execution is that, instead of executing a program with concrete values, we execute it with symbolic values, which effectively execute multiple values at one time.
- "**Symbolic" execution** uses **symbols** representing **a set of values** instead of concrete values

### Constraint solving

A key technology required for symbolic execution. Constraint solving is an approach to generating solutions for mathematical problems expressed as constraints over a set of objects. Each constraint to be solved has three parts:

1. a set of variables;
2. a domain for each variable (e.g. the integers, characters, or real numbers);
3. the set of the constraints.

Constraint solvers can be used to answer three types of questions:

1. Is there a solution to this problem?
2. If "yes" to 1, what is a solution to the problem?
3. Is one constraint entailed by another?

### Symbolic states

Constraints can be used to assign symbolic values to program variables. By symbolic, we mean values that represent more than one value.

- When we perform standard software testing, we provide a program with concrete inputs, and those inputs are executed.
    - Exploring the entire input space with concrete values is impossible for even the most trivial examples.

The key idea behind symbolic execution is to replace the concrete state with a symbolic state, in which the variables map to symbolic values, and to perform calculations on those symbolic values.

- By doing this, a program can be executed for many inputs at one time.

A common property for symbolic execution tools is to **check whether functions can return null**.

- To check whether the pointer can possibly be null, we have to check whether for at least one path, whether the final path constraint conjoined with the constraint is satisfiable.
    - If it is satisfiable, there is at least one execution of the program that results in a null pointer.

Symbolic execution can find considerably more subtle faults

- subtle faults are difficult to detect using dynamic software testing, and are where symbolic execution can be useful.

## Symbolic test oracles

Symbolic execution is only a way to explore test inputs for a program, but it cannot tell us whether that program meets its requirements.

- For this, we need the equivalent of a test oracle for symbolic execution. Most programs do not contain useful assertions that can be used.
    - Standard test oracle solutions cannot necessarily be applied directly, because they are designed for concrete examples, not symbolic examples.

Despite this, some of these solutions would work well.

1. For example, if we have a golden program, then we can execute a path the  using symbolic execution, and compare the final path constraint to the path constraint generated when executing the same symbolic input on the golden program.
2. Also, the idea of metamorphic oracles can be applied to symbolic execution
    a. for some programs, a specific symbolic value may return a particular output.

In most cases, symbolic execution is used to look for generic faults

- such as accessing arrays out of bounds, dividing by zero, returning null pointers, or certain security violations.
- These can be checked for any program using a generic “oracle”, and empirical evaluation shows that they work very effectively.

## Test Input Generation

An important part of symbolic execution is test input generation

- test inputs can be generated by asking the constraint solver for a solution to the path constraint.
- If our symbolic execution tool finds that the program can indeed return a null pointer, then we can also ask for a concrete test input that produces this case.
    - This provides programmers with a value concrete case for debugging, which are much easier for humans to reason about than symbolic traces.

## Limitation

### Constraint solving

Symbolically executing all inputs on a path is expensive due to the cost of the underlying constraint solving. **High overhead! - Long path with complicated constraints**

- Current evaluations put the cost of symbolically executing a path at about 80 times that of concrete execution.
    - This means that even a small set of unit tests covering a set of paths that take one minute to execute would take over an hour to symbolically execute.

Two solutions significantly **reduce the execution cost of symbolic execution**:

1. Removing unnecessary constraints
    a. Each branch in a program generally depends on a small number of variables (1 or 2), to solve a constraint at the branch (to fork the symbolic execution process), we can **eliminate those constraints that are not relevant**.
2. Caching solutions
    a. Branches in programs tend to have lots of constraints that are similar to each other. When solutions are required; e.g. generating a test input; the symbolic execution tool can cache solutions and try to reuse them later.

However, **the problem will persist**, as reasoning over an entire collection of inputs will always take much longer than a single input.

**Path explosion**

If a symbolic execution tool probes every potential path in a program, the path count skyrockets exponentially with each additional program branch.

Much research has gone into mitigating the path explosion problem, sacrificing completeness for scalability.

- Most of this research looks for heuristics for executing only a small subset of possible paths while minimizing the impact this has on fault finding.
- **With each if statement, the number of possible branches might double.**
- **Problem is worse with loops.**
- Some typical solutions
    a. Coverage-based search
        - iteratively chooses the path with the highest number of unexplored statements/branches to achieve branch coverage rather than path coverage, or the paths with code that was just added to the program.
    b. Random path search
        - works surprisingly well, but not as systematically as coverage-based search.

**Loops**

Loops further exacerbates the path explosion problem by giving us a potentially infinite number of paths.

Moreover, even with testing rules like the 0-1-many to choose limited paths in a loop, certain paths may need a precise number of loop executions.

- Symbolic execution tools lack the advanced reasoning to figure this out and can blindly proceed, causing them to get "stuck."

**Unsolvable paths**

those paths that cannot be symbolically executed, effectively halting exploration of the path.

Paths are typically unsolvable for two reasons:

1. Unsolvable constraints
    a. a branch or instruction contains a statement that is outside of the scope of the constraint solver.
2. External calls
    a. if a program contains a call to an external function, such as a library, or a function that reads user input, then it cannot reason about this symbolically, as it does not have access to the source code.
        i. Thus, the symbolic execution algorithm must either halt, or approximate what the answer will be for the output.

The **KLEE symbolic execution tool has its own model** of the Linux system library, which it uses to get around this problem; however, this is **not a generalisable solution**.

### Tools

1. Java PathFinder
    a. A model checker and symbolic execution tool for Java, built and maintained at the NASA Ames Research Centre.
2. KLEE
    a. the most mature symbolic execution tool for LLVM (Low Level Virtual Machine), and many research projects have been built on KLEE since its release in 2008. KLEE is **still actively maintained**.
3. Sypy
    a. A symbolic execution tool for Python, which has not been systematically evaluated. It appears that the development of the **tool has shut down**.

# Week 10

## Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) merges random testing with symbolic execution, aiming to bypass the limitations of both methods in generating test inputs.

- a combination of dynamic execution and symbolic execution.
    a. A DSE tool runs a concrete input on a program, and simultaneously performs symbolic execution on the path that this input executes.
    b. Then uses the path constraint generated by the symbolic execution to generate a new test.

DSE is used for dynamic test input generation.

- It uses symbolic execution, but the **approach** to test input generation is **different** to that of symbolic execution.

The difference between symbolic execution and DSE is two things:

1. In DSE, a real input is run alongside a symbolic one, rather than creating the test post symbolic path analysis. The initial input is selected randomly or by chance.
2. Path constraints are modified to generate a new test after each path is executed.

Advantages

1. Compared with random testing, it is not so random.
    a. While the first test case is random, using the program's control-flow to explore the input space provides a much better coverage of the program structure than with random testing.
2. Compared with symbolic execution
    a. If DSE hits an unsolvable path, it adopts the actual values for unsolvable variables, incorporating them into the symbolic state. While this somewhat simplifies the symbolic values, it enables the execution to proceed.
    b. It opens up to new search strategies to deal with the path explosion problem.
        i. For example, for their SAGE DSE tool, Microsoft researchers proposed a technique known as generational search. In generational search, one symbolically executed path is used to generate multiple new tests.
3. This generational search algorithm could allow DSE to **scale much better than** using a **more complete algorithm**, such as depth-first or breadth-first search.

Disadvantages

1. DSE still suffers from all of the same limitations as symbolic execution
    a. path explosion, expensive constraint solving, loops, and unsolvable paths.
        i. However, the **use of concrete information during execution** helps to mitigate all of these at some level, significantly **improving the scalability**.
2. Divergence.
    a. Not exist in symbolic execution.
    b. DSE algorithms use a concrete input to determine the next path and don't fork at each branch; they just gather the symbolic values of the ongoing path. If a generated test leads to an unexpected path, it's called a divergence, stemming from constraint solvers' limitations.

## Tools

1. Microsoft SAGE
    a. The most successful DSE tool to date
    b. SAGE is not available for download, however, it has been used successfully internally at Microsoft since 2008, finding over one third of all security vulnerabilities found in Windows 7.
2. S^(2)e

    a. an open-source DSE tool for the LLVM, built on KLEE, which contains a sophisticated approach for generating sequences of method calls to find vulnerabilities in programs.
3. CREST and jCUTE
4. open-source DSE tools for C and Java respectively, primarily designed to allow researchers to modify and experiment with DSE.