# An Introduction to Engineering Methods

## Software Engineering Methods

We will define engineering methods, somewhat loosely, as follows.

**Definition 1** *An* Engineering Method *is a set of activities, notations, methods or mathematical tools designed to measure, monitor and control a specific set of system properties.*

By a *system property* we will mean specific predicates that hold, or should hold, of the system as a whole. Whether or not the system meets specific *performance* or *reliability* measures are good examples here.

Some properties such as whether or not a system is *Safe* or *Usable* are not easy to quantify and must often be reduced to those that are quantifiable if they are to be measured and monitored during development.

## . . . And Now for a Walk in the Quality Assurance Forrest

In this lecture we will do this by exploring three key ideas:

(i) *Quality* and how do we break it down so that we can measure, monitor and control "*quality*";

(ii) *Assurance*, or how confident are we that the final program meets our requirements and how do we guarantee that the final program *will* meet our requirements; and

(iii) *Engineering Methods*, or the idea that we can employ certain techniques and tools to give us confidence that we have met our system requirements.

## Quality for Software Engineers

*Ideally, for engineering we would like to be able to monitor, measure, evaluate and control quality!. . . But what is quality?*

Quality is considered simply too vague a concept to "*engineer*". For engineering purposes we need to *measure*, *monitor* and *control* quality and so more quantifiable notions of quality are required.

## Quality Models

Typically, a *quality model* is used to give a more precise and (where possible) measurable notion of software quality.

Quality models typically decompose the concept of quality into a *hierarchy* of characteristics each of which contribute to the overall quality of a piece of software.

## The Concept of Quality

Lets begin with a relatively abstract characterisation of *quality*.

- satisfy *explicit* requirements, that is, it should be correct, complete and consistent with respect to explicit requirements;
- adhere to internal (organisational or project) and external standards imposed on the project, for example IEC61508 (safety), Rainbow Standards (US Security), IEEE standards, internal company standards;
- conform to implicit *quality* requirements which are requirements for performance, reliability, usability, extendibility, safety and security and typically capture what we think of as the attributes of quality.

**Note:** Some factors may require more attention than others because of the problem domain, customer requirements or business necessity.

## Quality According to ISO-9126

Next, we’ll break down quality to give more detailed *Quality Model*. AN example is ISO-9126 where quality is characterised by the following *hierarchy of quality attributes*.

| **Characteristics** | **Sub-characteristics** |
|---|---|
| Functionality | Suitability |
| | Accuracy |
| | Interoperability |
| | Security |
| Reliability | Maturity |
| | Fault Tolerance |
| | Recoverability |
| Usability | Understandability |
| | Learnability |
| | Operability |
| | Attractiveness |
| Effi ciency | Time Behaviour |
| | Resource Utilisation |
| Maintainability | Analysability |
| | Changeability |
| | Stability |
| | Testability |
| Portability | Adaptability |
| | Installability |
| | Co-existence |
| | Replaceability |

ISO-9126 Quality Attributes and Their Sub- attributes.

## Quality Assurance

*Can we guarantee that our program has ”quality”*?

- It is difficult, if not impossible, to *guarantee absolutely* that we have met all of our quality requirements.
- Rather the aim is to provide a high level of *assurance*—a high degree of confidence—that the resulting system will meet the needs for which it was built.
- Meeting this need means ensuring that the system meets a range of requirements stemming from the user and the operational environment.
- An important factor that distinguishes engineering from ad hoc development is the ability to exert control over the level of assurance achieved in a project to suit the purpose of the project.

## Quality Assurance

In practice, exerting this kind of control over quality and quality attributes requires that we build quality and its attributes *into the system* starting at requirements stages.

*All the evidence that we have points to a situation where we cannot simply fix up our software* **post hoc** *and add in quality*!

## Which Engineering Methods?

Engineering methods are aimed at ”building in” key properties or quality attributes into the software.

The engineering methods that we consider here are those dealing with:

- *Correctness, Completeness, Consistency* which is what we usually aim for as a minimum level of quality. A combination of audits, technical reviews and execution based testing is typically used to assure the three C’s.
- *Software Reliability Engineering*, is a set of activities and analytical methods that relies on testing, and other methods to get data for assessing and predicting properties such as mean time to failure or failures per unit time;
- *Safety Engineering*, which is a set of activities and analytical tools to build systems with the potential to accidently kill or injure people.
- *Performance Engineering*, is a set of activities and analytical methods aimed building systems with specifi c timing or resource usage requirements.

## Basics

While the various methods that we will study have evolved separately there is some commonality between the methods and, apart from testing, this is how we will approach the study of our engineering methods.

Each method has:

- a set of activities that are to be performed during various lifecycle phases;
- a supporting modelling theory for evaluating the artifacts produced by the development process; and
- guidelines and techniques for building systems to meet the specific quality attributes or special requirements.

## An Engineering Method for Reliability

A first example is reliability engineering.

**Activities** are given by Software Reliability Engineering (SRE) methods:

- Setting up "*system*" and sub-system test and measurement methods;
- Recording and interpreting results;
- Improving process and product according to results.

**Modelling Theory** is based on reliability growth models for hardware and software:

- Based on the theory of probability, Markov Chains and Poisson and Binomial distributions; and
- Basic execution time and Poisson models.

## An Engineering Method for Reliability

**Product guidelines and Reliability Techniques**
is based on the fault-tolerant design.

- Understanding faults and how they can manifest in the system including benign failures and byzantine failures;
- Redundant designs such as *Triple Mode Redundancy* and determining faults through voting and byzantine agreement;

## An Engineering Method for Performance

A second example is performance engineering;

**Activities** are given by Software Performance Engineering (SPE) methods:

- Setting up “*system*” and sub-system test and measurement methods;
- Recording and interpreting results;
- Improving process and product according to results.

**Modelling Theory** is based on Execution graphs, timing diagrams and timing variance.

## An Engineering Method for Performance

**Product guidelines and Reliability Techniques**

is still much more of a trial and error approach but the theory allows us to model and improve architectures, designs and other non-executable artifacts through a “*best guess*” process.

## Safety

Safety is a little different because safety engineering currently relies heavily on process and the ability to integrate a *Safety Life-cycle* with a *Development Life-cycle*.

**Activities** are based on constantly assessing and reassing artifacts for safety using a variety of techniques.

**Modelling Theory** is based around various hazard analyses and risk assessments.

**Product guidelines and techniques** typically go back to the reliability and fault tolerance of the *safety related system functions*.

## Testing

*What can we test for*?

- *Correctness*, *Completeness* and *Consistency* using unit tests, integration tests and acceptance tests.
- $\alpha$ and $\beta$ testing.
- Recovery testing – force the system to fail in various ways and then verify that correct recovery occurs.
- Security testing – aims to test that the security protections built into the system do indeed protect the system from attack.
- Performance testing – aims to assess the performance of the system under normal or abnormal loads, perhaps by building up an operational profile of the system performance.

## Testing

In particular we will use testing methods for assuring correctness, completeness and consistency, *and* for measuring the reliability and performance aspects of our systems.

Testing underlays most of what we will do and that is why we will attack testing first.

# An Introduction to Software Testing

## Software Testing?

Testing, at least in the context of these notes, means *executing* a program in order to find failures - departures of the program behaviour from specifications or intentions. In fact, that is all we can do with software testing!

- We cannot *prove* that a program meets its specification or does its job by testing alone – there are simply too many inputs and too many cases to cover.
- Unless we undertake *exhaustive* testing, that is, try every input possible on every path through program, then we also cannot guarantee the absence of bugs by testing alone.
- **BUT** without testing a program we have no confidence at all that the program will do its job on the intended platform in the intended environment at all.

## Various Perspectives on Software Testing

- Establishing confidence that a program does what it is supposed to do (W. Hetzel, *Program Test Methods*, Prentice-Hall)

- The process of executing a program with the intent of finding errors (G.J. Myers, *The Art of Software Testing*, John-Wiley)

- The process of analysing a software item to detect the difference between existing and required conditions (that is, bugs) and to evaluate the features of the software item (IEEE Standard for Software Test Documentation, IEEE Std 829-1983)

- The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component (IEEE Standard Glossary of Software Engineering Terminology, IEEE Std 610.12-1990)

## Our Perspective on Software Testing

From our perspective, software testing gives us an important set of methods that can be used to *assure* the quality of software systems.

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

(i) The correctness of software;

(ii) The performance of software under various conditions;

(iii) The robustness of software, that is, its ability to handle erroneous inputs and unanticipated conditions;

(iv) The usability of software under various conditions;

(v) The reliability, availability, survivability or other dependability measures of software; or

(vi) Installability and other facets of a software release.

## Remarks on Software Testing

Before going further some remarks need to be made.

(i) Testing is based on *executing* a program, or its components. Therefore, testing can only be done when some parts (at least) of the system have been built.

(ii) Some authors include V&V activities such as audits, technical reviews, inspections and walk-throughs as part of testing.

We take the converse stance and view testing as part of quality and its assurance.

## The Language of Failures, Faults, and Errors

- **Failure:** The inability of the system or a component to perform its required functions within specified performance requirements.
- **Fault:** An incorrect step, process, or data definition in a computer program. Faults are the source of failures – a fault in the program triggers a failure under the right circumstances.

  In normal language faults are usually referred to as “*errors*” or “*bugs*”.
- **Error:** The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

## The Language of Failures, Faults and Errors

Consider the following simple program whose specification is that for any integer $n$, $\mathrm{square}(\mathrm{n}) = \mathrm{n}^2$ and the following program:

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

Executing `square(3)` results in a *failure* because our specification demands that the computed answer should be $9$. The *fault* leading to failure occurs in the statement `return x*2` and the *error* between computed results and specified results is $3$.

**Note:** executing `square(2)` would not have resulted in a failure.

## Experiencing Failures, Detecting Faults Removing Faults

**Observation** *In testing we can only ever detect failures*! Our ability to find and remove *faults* in testing is closely tied to our ability to detect failures.

We would normally undertake the following three steps when testing and debugging the software component.

- Detect system failures through testing;
- Determine the faults leading to those failures; and
- Repair and remove the faults leading to the failures.

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). Consequently, the process is also subject to quality assurance activities.

## Programs

We will adopt the following view of programs.

**Definition 2** A program $P$ is a relation between *inputs* and *outputs.*

The relation can be *deterministic*:

- For every input $x$ there exists a unique output $y$ such that $P$ executed with input $x$ computes $y$.

The relation can be *non-deterministic* as can be the case in programs with multiple executing threads:

- For every input $x$ there exists a number of possible outputs $y$ such that $P$ executed with input $x$ computes $y$.

We will deal only with deterministic programs.

## Programs

Programs may *terminate*!

- In the case of testing a program that terminates we would expect that $P$ executed on the input $x$ would terminate, and deliver an output $y$ and that $y$ meets the specification of $P$.

## Programs

Programs may not terminate!

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

  (i) generate observable outputs, in which case we need to establish that the required sequences of outputs are produced; or

  (ii) it may not generate any observable outputs in which case we are interested in the sequence of internal state changes that the component undergoes. This is better thought of in terms of state machines.

  *(see Object Oriented Testing later).*

## Inputs

We will need to think carefully about *inputs*. Inputs to the program or component need not come from program parameters or read statements alone. They can come from all of the following sources.

- Inputs passed in as parameters;
- Entered by the user via the interface;
- Read from files;
- 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 fi les in the fi le system;
  - Saved data from interrupts and interrupt handlers.

## Inputs

In general, the inputs to a program or a program component are stored in *program variables*. A program variable may be:

- A variable declared in a program as in the *C* declarations

```
 int base;
char s[];
```

**Remark 3** Variables that are inputs to a component under test

(i) can be defined in some enclosing scope, for example, they are declared global to the program

(ii) can be *structured data* such as linked lists, files or trees, as well as atomic data such as *integers* and *floating point numbers.*

## Inputs

- A reference or a value parameter as in the *C* function declaration

```
      int P(int *power, int base) {
         ... do stuff ...
     }
```

- Constants declared in an enclosing scope of the component, for example,

```
#define PI 3.14159

double circumference(double radius)
{
    return 2*PI*radius;
}
```

- Resulting from a read statement or similar interaction with the environment, for example,

```
   scanf(‘‘%d\n’’, x);
```

## Input Domains

The *input domain* to a program is the set of values that are accepted by the program as inputs. **Note** that this does not mean that all of the values in the input domain are valid inputs.

Input domains are derived from two main sources.

(i) The specification of the program – recall our view of the program as relation between the set of all inputs and the set of outputs.

(ii) Each variable that is an input to the program under test has a set of values associated with it, that is, the values associated with the *data type* of the variable.

## Input Domains

*Normally* the values to test the program are drawn from the input domain of the program.

We say *normally* because for some cases the inputs to the program may require:

- sequences of values in the input domain, for example, when testing object oriented programs using state machines a sequence of values may be needed to force a specifi c state of an object; or
- sets of values that force a specifi c condition, for example, taking a large data set to test through-put in a program.

# Outputs

## The Program is Deterministic

| | |
|---|---|
| Returns an explicit value | The specification determines the unique value returned by the program. |
| Does not return an explicit value | The specification determines the unique final state of the program. |

## The Program is Non-deterministic

| | |
|---|---|
| Returns an explicit value | The specification determines a *set* of acceptable values that can be returned by the program. |
| Does not return an explicit value | The specification determines a *set* of acceptable values that can be returned by the program. |

## Test Cases

Ultimately, testing comes down to the selecting and executing *Test Cases*. A test case for a specific component consists of three essential pieces of information:

(i) A set of test inputs;

(ii) The expected results when the inputs are executed; and

(iii) The execution conditions or environments in which the inputs are to be executed.

## Specification Based Testing?

Testing uses the system, program or component specification to determine the expected outputs when choosing test cases.

But there are many occasions when testing is required for programs with no clear specification. In this case, the job of the tester is to gather as much information about the expected behaviour of the program as possible using:

- advertising;
- user manuals; and
- documentation, design notes or other development notes that are available.

## Two Key Testing Strategies

Both of the following test case selection strategies are specification based testing strategies.

**Black-box Testing** where test cases are derived from the functional specification of the system; and

**White-box Testing** where test cases are derived from the internal design specifications or actual code (sometimes referred to as *Glass-box*).

## Black Box Testing

Black box test cases are generated from the functional specification of the system without reference to the internal structure of the system.

- The component, program, or system, is considered as a *Black Box*.
- Testing is only concerned with functionality and features of the system but not its internal operations.
- The internal operations of the system may not be known at the time of designing test cases.

## Features of Black Box Testing

### Advantages

1. The design of *test cases* is independent of the detailed design or the software code.

   Consequently, generating the test cases and coding the system can be done in parallel.

2. Black box testing methods can test for *missing functions*, that is, functions specified but not implemented.

### Disadvantages

3. Black box testing cannot test for functions, that is, functions implemented but not specified.

## White Box Testing

White box test cases are generated from the specification of the component and the internal design or code of the system. Of course this means that testers need access to the most recent design documents.

**Advantages**

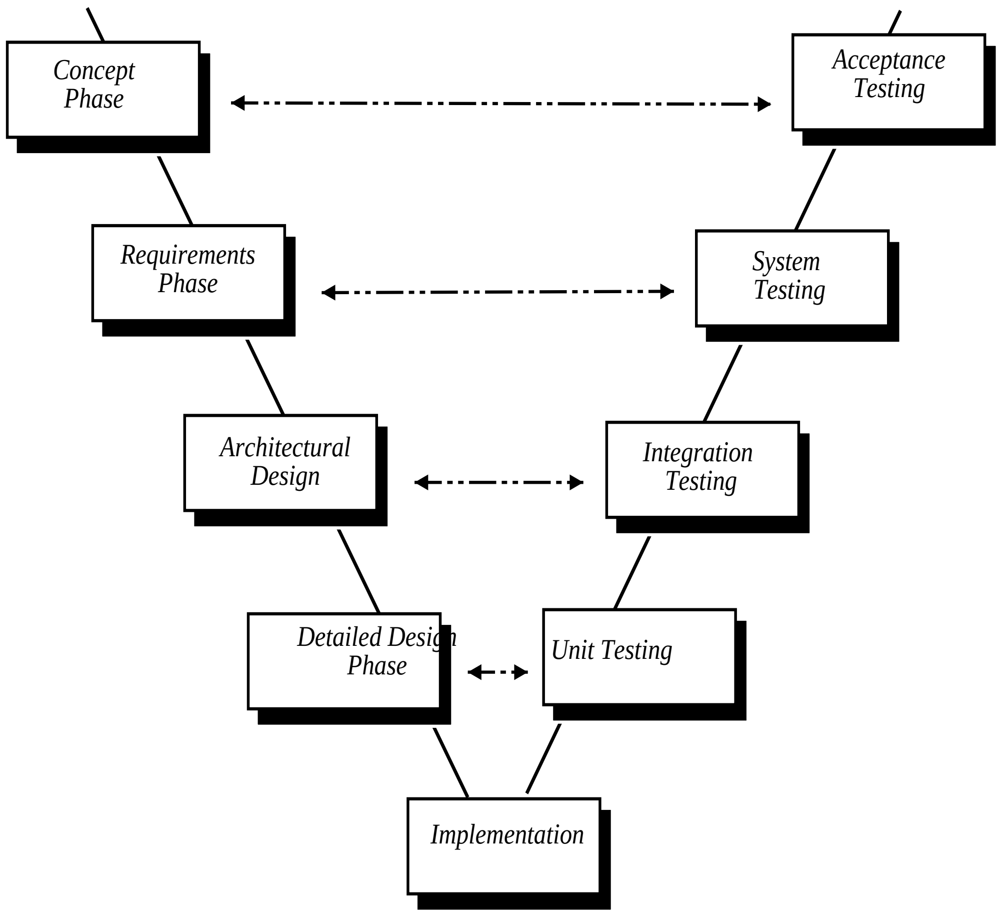
1. White box testing can test extra functions and can test for extra functions.

**Disadvantages**

2. White box testing methods cannot test for missing functions.
3. The selection of test cases can only be carried out after the coding the module(s) to be tested.

## Different Levels of Testing: The "V" Model

There are a number of different types and levels of testing. The following *V* model is often used to show where different forms of testing fit into the overall software engineering process.



## Unit Testing

Unit testing is the process of testing the individual components or collections of components, that is, methods, object classes, subprograms or procedures, of a program.

- When unit testing, the "*input domains*" of the units are consider1d and the remainder of the system is ignored.
- Units tests are usually, but not always, tested against detailed design specifications.
- Unit testing typically involves the construction of test *stubs* or *drivers* in order to simulate a unit's environment.
- Unit tests can be *Black Box* or *White Box* (see below).

## Integration Testing

Integration testing deals with testing collections of components that have each bee tested prior to integration. The aim of integration testing is to test the interfaces and communication between components.

- Typically, the focus is on the subset of the input domain to the integrated component that will exercise the interfaces and communication between components.
- Performance and reliability testing can also be performed at this level.
- Integration testing is often performed against various stages of intermediate design as well as the software architecture.
- Some examples of interfaces between components that are part of an integration test include:
  - Specifi c sequences of method or function calls to establish a specifi c state within a component;
  - Messages, packets and event flows between components;
  - Data flows between components and components and external data-bases, networks and fi les.
- Again, *stubs* and *drivers* are often required to simulate an integrated component's environment and provide test inputs and record outputs.

## System and Acceptance Testing

System testing involves testing the entire integrated set of components that make up a deliverable. System testing is usually done against system requirements specifications.

Acceptance testing is done by the clients or procurers of the software. It is the testing done by a client or some other agency to determine if the software, or software component should be accepted or not.

## Test Planning and Execution

As always a systematic approach to testing gives us the confidence that our system goals are being met.

A *Testing Process* for a specific level of testing (System, Integration, Unit) includes the following information.

1. Defi ne the testing objectives;
2. Design test cases to meet those objectives;
3. Generate the test cases using testing strategies that meet your testing objectives;
4. Determine the expected output for each of the test cases;
5. Execute the test cases;
6. Analyse the test results i.e., compare the actual output with the expected output;

Normally testing continues until some *Test Coverage* goals or some *Dependability* goals have been met.

## Test Execution – Instrumentation

It can be difficult for test cases to create the conditions necessary for comprehensively testing a program or system.

You may need to ask development team to add test-points to the system.

A test-point is a permanent point in the product that enables a tester to:

- externally interrogate and set the value of a variable;
- perform one or more specific actions, as indicated by the value of the variable; or
- does nothing if the variable is set to its default value.

## Test Execution – Instrumentation

Examples:

- To halt the system for testing recovery procedures;
- To introduce timing delays;
- To invoke a procedure supplied by the tester; and
- To generate an input/output error condition in a channel, controller or device to test recovery procedures.

## The Key Laws of Testing

**Knuth's Law** Testing can only be used to show the presence of errors, but never the absence or 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 principal** 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.

## References

1. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.

2. B. Beizer, *Software Testing Techniques*, 2nd ed., van Nostrand Reinhold, 1990.

3. E. Kit, *Software Testing in the Real World*, Addison-Wesley, 1995.

4. A. Endres and D. Rombach, *A Handbook of Software and Systems Engineering*, Addison-Wesley, 2003.

5. J. A. Whittaker, *How to Break Software: A Practical Guide to Testing*, Addison-Wesley, 2002.

# Black Box Testing

# or

# Functional Testing Strategies

## Basics

Black box testing considers the program as...

...a *Black Box*.

Testers are only concerned with the functions and features of the program as given by the specification, but are not concerned with the program's internal operations.

Test cases are selected based on the specification of the program.

The input domain is derived from the specification of the program.

## Basics

Common *Black Box Specification Based Testing* strategies include the following.

- Random testing
- Specification based methods, that is, methods that rely on an analysis of the specification to determine test cases;

  1. Equivalence Partitioning
  2. Boundary-value Analysis
  3. Error Guessing

## Random Testing

In random testing approaches the data used for test cases is randomly selected from the input domain of the program.

In this case, we do not have the overhead of having to analyse the input or output space of the program. However, we do need a good random number generator to help select “*points*” in the input space.

For a discussion of good random number generator, see, e.g., D.E. Knuth, *The Art of Computer Programming, vol. 2: Semi-numerical Algorithms*, 2nd Ed., Addison Wesley, 1981.

## Random Testing

One major application of random testing is to provide data for estimating software reliability. Test cases are randomly generated according to an *Operational Profile*, and data such as failure times are recorded. The data obtained from random testing can then be used to estimate reliability. No other testing method can be used in this way to estimate reliability.

The *Operational Profile* of a program in the input domain is the probability distribution of selecting points in the input domain in the case when the program, or system, is being used in actual operation.

However, any reliability predictions are incorrect if the operational profile is incorrect.

**Additional Reference**: R. Hamlet, “Random Testing”, In *Encyclopedia of Software Engineering*, J. Marciniak ed., pp. 970–978, Wiley, 1994.

## Specification Based Testing

The alternative to random testing is to generate test cases based on an analysis of the software/system specification.

### Specifications

We will make the following distinctions in this subject.

- Informal – typically a combination of everyday language and diagrams;
- Rigorous – typically using semi-formal graphical notations such as UML, OMT, OMG notation, Structured Analysis and Design notation (for example, Data-flow diagrams or structure charts);
- Formal methods – typically using mathematical notation or notations with a sound mathematical underpinning such as **Z**, VDM, **B**, CCS, CSP, LOTOS, Algebraic and Axiomatic methods.

In this subject we will often use *Informal* and *Rigorous* specification methods as the basis for testing, but sometimes we will logic and set theory and these are *Formal*.

## Equivalence Partitioning

The intuition behind equivalence partitioning is to divide the input space of the program into classes (sets really!) of related test cases. The sets are called as *Equivalence Classes*.

- Each equivalence class represents a *Condition* on the input, for example, the set of *valid* input data or the set of *invalid* input data.
- A test case taken from an equivalence class is representative of all of the test cases taken from that class.
- Equivalence partitioning significantly reduces the number of input condition to be tested by identifying classes of conditions that are equivalent to many other conditions.

The key question is – *What is an equivalence class*?

## Equivalence Partitioning
## Input Conditions

**Definition 4** *An* Input Condition *is a predicate over the input domain of a program that specifies the set of valid inputs to the program.*

Input conditions are typically used to partition the input domain into equivalence classes for the purpose of selecting inputs. Consider a modified version of the specification of the square program for returning unsigned integer results on 32 bit machines:

$$\forall\, \mathsf{x} \in \{0, \ldots, 65535\} \bullet \mathsf{square}(\mathsf{x}) = \mathsf{x} * \mathsf{x}.$$

- The input domain is the set of unsigned machine integers; and
- The sole input condition is $x \in \{0, \ldots, 65535\}$.

## Equivalence Partitioning

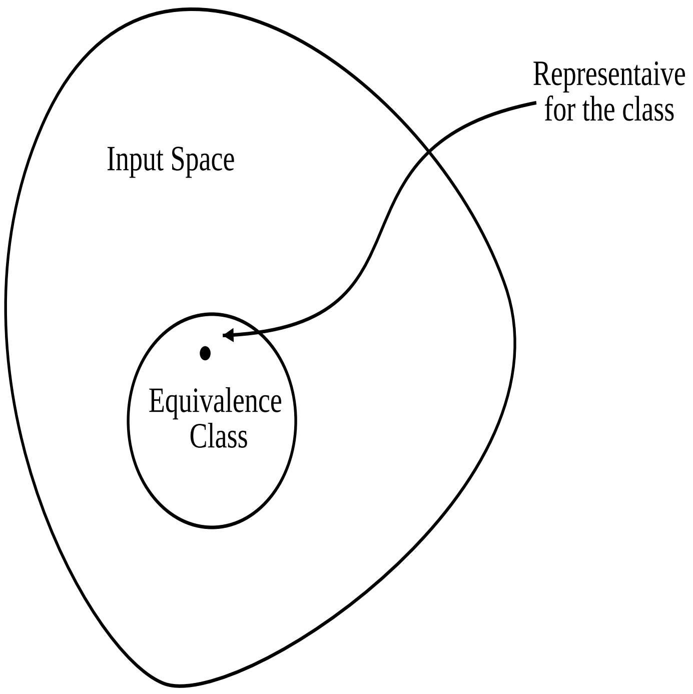
1. Equivalence partitioning is a systematic method for identifying sets of interesting classes of input conditions to be tested.
2. Each class is representative of (or covers) a large set of other possible tests.
3. Equivalence partitioning does **NOT** test combinations of input conditions.

We really expect the program or component to "*behave*" in much the same way for all elements of an equivalence class. Instead of writing test cases for all elements of the class, we can pick one representative test-case and infer the properties of the class from the test-case.

*Care is needed here in picking equivalence classes - we must ensure that all members of the class behave the same so that our inferences are valid*.

## Equivalence Partitioning

Intuitively equivalence partitioning relies on the following idea.



## Equivalence Partitioning

**Method:**

- The *aim* is to minimise the number of test cases required to cover the input domain. There are two distinct steps:

  1. Identify the equivalence classes (ECs);
  2. Identify the test cases.

## Identifying Equivalence Classes

### Guidelines

1. If an input condition specifies a range of values, identify one valid equivalence class and two invalid equivalence classes.

   **Example 5** *If we are given the range of values* $1 \ldots 99$ *then we require three equivalence classes:*

   - *The valid equivalence class* $\{1, \ldots, 99\}$*; and*
   - *Two invalid equivalence classes* $\{x \mid x < 1\}$ *and* $\{x \mid x > 99\}$*.*

## Identifying Equivalence Classes

2. If an input condition specifies a set of input values and each is handled differently, identify a valid equivalence class for each element of the set and one invalid equivalence class.

   **Example 6** *If the input is selected from a set of say $N$ items then we require $N+1$ equivalence classes:*

   - *One valid equivalence class for each element of the set $\{M_1\}, \ldots, \{M_N\}$; and*
   - *One invalid equivalence class for elements outside the set $\{x \mid x \notin \{M_1, \ldots, M_N\}\}$.*

3. If there is reason to believe that the program handles each valid input differently, then define one valid EC per valid input.

   **Example 7** *If the input is from a menu then we define one valid equivalence class for each menu item.*

## Identifying Equivalence Classes

4. If the input 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 *zero values*, and one for values $> N$.

5. If an input condition specifies a “must be” situation, identify one valid equivalence class and one invalid equivalence class.

   **Example 8** *The first character of an input* must be *a numeric character then we require two equivalence classes – a valid class*

   $$\{s \mid \text{the first character of } s \text{ is a numeric}\}$$

   *and one invalid class*

   $$\{s \mid \text{the first character of } s \text{ is not a numeric}\}$$

6. If elements in an equivalence class are handled differently by the program, then split the equivalence class into smaller equivalence classes.

## Identifying Test Cases

```
1. Assign a unique number to each EC.

2. while all valid ECs have not been
         covered by test cases do
3.           write a new test case covering as many
             of the uncovered ECs as possible.

4. while all invalid ECs have not been
         covered by test cases do
5.           write a test case that covers one,
             and only one, of the uncovered invalid ECs.
```

## Equivalence Partitioning: Example 1

Consider the following *informal specification* for the **Triangle Classification Program**.

(i) The program reads in three costive integer values from the standard input.

(ii) The three values are interpreted as representing the lengths of the sides of a triangle.

(iii) The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled.

## Equivalence Partitioning: Example 1

- *What is the input domain for the program*?
  The possibilities include:
  - the set of all possible input strings - we are testing for robustness and we want to know if incorrect strings are properly rejected;
  - the set of all possible machine integers, both positive and negative; or
  - the set of all possible positive integers.
- *What are the input conditions*?
  For the specification of the triangle classification program (in the slide above) we have the case of four valid types of input and (according to case (2) above) so we need four valid equivalences classes, one for each type of triangle, and one invalid class for a non-triangle.

**Note** that we are using clause 6. to determine these equivalence classes.

## Equivalence Partitioning: Example 1

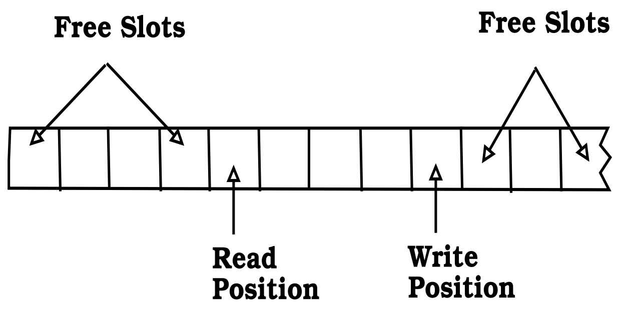
The resulting equivalence classes appear in the following table.

| **Equivalence class** | **Test Case** |
|---|---|
| scalene triangle | $\{(3, 5, 7), \ldots\}$ |
| isosceles triangle | $\{(2, 3, 3), \ldots\}$ |
| equilateral triangle | $\{(7, 7, 7), \ldots\}$ |
| right-angled triangle | $\{(3, 4, 5), \ldots\}$ |
| non-triangle | $\{(1, 1, 3), \ldots\}$ |
| non-positive input | $\{(-1, 0, 3), (-2,-3,-3),\ldots\}$ |

## Equivalence Partitioning: Example 2

Next consider a more complex example where we Black-Box test a *Ring Buffer* ADT. The informal specifi cation is as follows.

1. The ring buffer consists of the following operations to store and retrieve messages in a distributed system.
   (i) The initBuffer operation takes no arguments and creates a new empty ring buffer.
   (ii) The writeBuffer operation accepts a ring buffer and a message and updates the buffer with the message inserted at the next available writing slot.
   (iii) The readBuffer operation takes a ring buffer and returns the element in the current writing position.
2. The ring buffer must be large enough to store up to 100 messages.
3. The reader and the writer must not access the same slot simultaneously. The reader must always read from one or more slots behind the slot currently being written to.
4. The key condition that must be maintained is that writeBuffer must not write to a slot that has not been read. The *free slots* in the ring buffer are those after the current writing slot but before the slot currently being read.



## Equivalence Partitioning: Example 2

```
int writeBuffer( struct Buffer *rb,
                 struct Message *msg)
{
  int rd, wr, fs;

  rd = rb -> read;
  wr = rb -> write;
  fs = rb -> freeSlots;

  if (fs == 0)
      return -1;
  else {
      rb -> contents[wr] = *msg;
      rb -> write  = (wr + 1) % BUFFER_SIZE;
      rb -> freeSlots--;

      return 1;
  };
};
```

## Equivalence Partitioning: Example 2

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>

#include "message.h"

#define BUFFER_SIZE   100

struct Buffer
  {struct Message contents[BUFFER_SIZE];
          int     read;
          int     write;
          int     freeSlots;
          int     status;
  };
```

## Equivalence Partitioning: Example 2

Here is one way of attacking the testing of write-Buffer using black box methods.

**Step 1** Let the set of ring buffers be denoted by $\mathcal{RB}$ and the set of messages by $\mathcal{M}$. From clause 1(i) the input domain to the writeBuffer is $\mathcal{RB} \times \mathcal{M}$.

**Step 2** We use the detailed design (code) to determine that $\mathcal{RB}$ is actually a product of an array `content` of up to 100 messages, an integer `read` for the reading position, and an integer `write` for the writing position, an integer `freeSlots` giving the number of free slots and an integer `status`.

Let the set of arrays of messages be $\mathcal{A}$.We conclude that a possible input domain is

$$\mathcal{A} \times \text{int} \times \text{int} \times \text{int} \times \text{int}.$$

## Equivalence Partitioning: Example 2

The input conditions so far are:

- for any input array $a \in \mathcal{A}$, $0 \leq \textit{length}(\mathcal{A}) \leq 100$;
- for any input array $\texttt{read} \neq \texttt{write}$;
- for any input array $0 \leq \texttt{freeSlots} \leq 100$ and $100 - (\texttt{write} - \texttt{read} \bmod 100) = \texttt{freeSlots}$;

Now combine the input conditions onto the fields (technically "*coordinates*") of set $\mathcal{RB}$ of ring buffers.

## Equivalence Partitioning: Example 2

The resulting equivalence classes are as follows.

1. A valid equivalence class with

$$\begin{aligned} EC_1 = \{\, rb \mid\ & \mathit{length}(\mathit{rb.contents}) \leq 100 \wedge \\ & \mathit{rb.read} \neq \mathit{rb.write} \wedge \\ & \mathit{rb.freeSlots} = 100 - \mathit{fs} \} \end{aligned}$$

   where $\mathit{fs} = (\mathit{rb.read} - \mathit{rb.write}$ mod $100)$.

2. Invalid equivalence classes are (one for each input condition):

   (i) $$\begin{aligned} EC_2 = \{\, rb \mid\ & \mathit{length}(\mathit{rb.contents}) > 100 \wedge \\ & \mathit{rb.read} \neq \mathit{rb.write} \wedge \\ & \mathit{rb.freeSlots} = 100 - \mathit{fs} \} \end{aligned}$$

   (ii) $$\begin{aligned} EC_3 = \{\, rb \mid\ & \mathit{length}(\mathit{rb.contents}) \leq 100 \wedge \\ & \mathit{rb.read} = \mathit{rb.write} \wedge \\ & \mathit{rb.freeSlots} = 100 - \mathit{fs} \} \end{aligned}$$

   (iii) $$\begin{aligned} EC_4 = \{\, rb \mid\ & \mathit{length}(\mathit{rb.contents}) \leq 100 \wedge \\ & \mathit{rb.read} \neq \mathit{rb.write} \wedge \\ & \mathit{rb.freeSlots} \neq 100 - \mathit{fs} \} \end{aligned}$$

**Exercise 9** *What will the test cases be?*

## Boundary-value Analysis

The intuition in Boundary Value Analysis is to select test cases to exercise the *boundary conditions* of a program and to see how well the program handles the boundary conditions

**Definition 10** Boundary Conditions *are those situations that are directly on, above, and beneath the edges of input equivalence classes and output equivalence classes.*

In boundary value analysis, test cases that explore the boundary conditions have a more important role that test cases that do not.

## Boundary-value Analysis vs Equivalence Partitioning

Boundary value analysis is both a refinement of equivalence partitioning and a variant of equivalence partitioning.

- Boundary-value analysis explores situations on and around the edges of the equivalence partitioning – boundaries are always a good place to look for defects (faults or failures).
- Boundary-value analysis requires one or more test cases be selected from the edge of the equivalence class or close to the edge of the equivalence class whereas equivalence partitioning requires that any element in the equivalence class will do.
- Boundary-value analysis also requires that test cases be derived from the output conditions and this is different to equivalence partitioning where only the input space is considered.

## Boundary-value Analysis Guidelines

1. If an input condition specifies a range of values, then construct valid test cases for the ends of the range, and invalid input test cases for situations just beyond the ends of the range.
2. If an input condition specifies a number of values, construct test cases for the minimum and maximum number of values and one beneath and beyond these values.
3. If an output condition specifies a range of values, then construct valid test cases for the ends of the output range, and invalid input test cases for situations just beyond the ends of the output range.
4. If an output condition specifies a number of values, construct test cases for the minimum and maximum number of values and one beneath and beyond these values.

## Boundary-value Analysis Guidelines

5. If the input or output of a program is an ordered set (e.g., a sequential file, linear list, table), focus attention on the first and last elements of the set.

6. Use your intelligence to search for other boundary conditions.

## Boundary-value Analysis: Remarks

- Boundary value is not as simple as it sounds – boundary conditions may be subtle and difficult to identify;

- Does NOT test combinations of input conditions.

## Boundary Value Analysis: Example 1

Now lets, change the specification of the triangle classification program as follows.

(i) The program reads in three positive *floating point* numbers from the standard input.

(ii) The three values are interpreted as representing the lengths of the sides of a triangle.

(iii) The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right-angled.

## Boundary-value Analysis: Example 1

We have the same equivalence classes as identified above. In addition we now have the following possible boundary conditions.

1. Given sides $(A, B, C)$ for a *scalene* triangle the sum of any two sides is greater than the third and so we have boundary conditions $A + B > C$, $B + C > A$ and $A + C > B$. For the $A + B > C$ case we can explore the boundary using the following test cases.

   $$(1, 2, 3),\ (1, 2, 2.999),\ (1, 2, 3.001), \ldots$$

2. Given sides $(A, B, C)$ for a *isosceles* triangle two sides must be equal and so we have boundary conditions $A = B$, $B = C$ or $A = C$. For the $A = C$ case we can explore this boundary using the following test cases.

   $$(2, 2, 3),\ (2, 1.999, 3),\ (2, 2.001, 3), \ldots$$

3. Continuing in the same way for an *equilateral* triangle the sides must all be of equal length and we have only one boundary where $A = B = C$ and we can explore this boundary using the following test cases.

   $$(3, 3, 3),\ (3, 2.999, 3),\ (3, 3.001, 3), \ldots$$

## Boundary-value Analysis

4. For *right-angled triangles* we must have $A^2 + B^2 = C^2$ and this gives us a boundary which can be explored using the following test cases.

   $$(3, 4, 5),\ (3, 4, 4, 999),\ (3, 4, 5.001), \ldots$$

5. For non-triangles we have similar boundaries to those above.

   $$(1, 2, 3),\ (1, 2, 2.999),\ (1, 2, 3.001), \ldots$$

6. For non-positive input we have

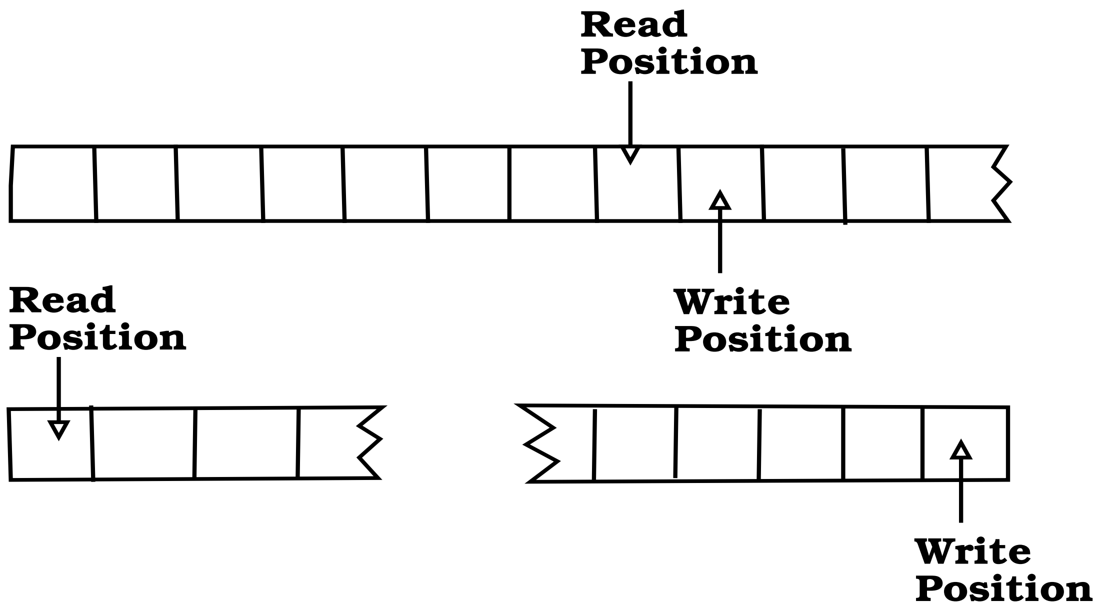
   $$(0, 1, 2),\ (-0.001, 1, 2),\ (0.001, 1, 2), \ldots$$

## Boundary Value Analysis: Example 2

Consider the writeBuffer example above and recall the equivalence classes that we derived earlier.

The main boundaries occur at:

- $\texttt{rb.read} = rb.write - 1 mod 100$
- $length(rb) = 100$



## Error Guessing Approaches

- Error guessing is an *ad hoc* approach based on intuition and experience.
- Identify test cases that are considered likely to expose errors.
- Make a list 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.

Examples include test cases for empty or null lists and strings, zero instances/occurrences, blanks or null characters in strings, and negative numbers.

# White Box

# or

# Structural Testing Strategies

## White Box Testing Strategies

In white box testing the aim is to use the structure of the code to select test cases. Doing this gives us a range of methods that can be used to uncover errors.

- Static Analysis - Most specifically data fbw analysis.
- Coverage Based Testing - Selecting test cases based on various criteria for "*covering*" the code:

  (i) Path coverage;

  (ii) Branch coverage;

  (iii) Condition coverage;

  (iv) Statement coverage;

- Domain Based Testing

## Control Flow Graphs

First we need *Control Flow Graphs* (CFG).

A *Control Flow Graph* is a graphical representation of the control structure of a program.

**Note** the control fbw graph of a module can be restricted to have one entry node and one exit node. Put another way, the control fbw graph of a program captures the various ways in which a program can execute.

*For differences between control flow graph and flowchart, see [Beizer, 1990, p.63]*

## Control Flow Graphs

- A node in a CFG represents a program statement;
- An edge in the CFG represents the ability of a program to fbw from its current statement to the statement at the other end of the edge;
- If an edge is associated with a conditional statement, label the edge with the conditional's value, either True or False.

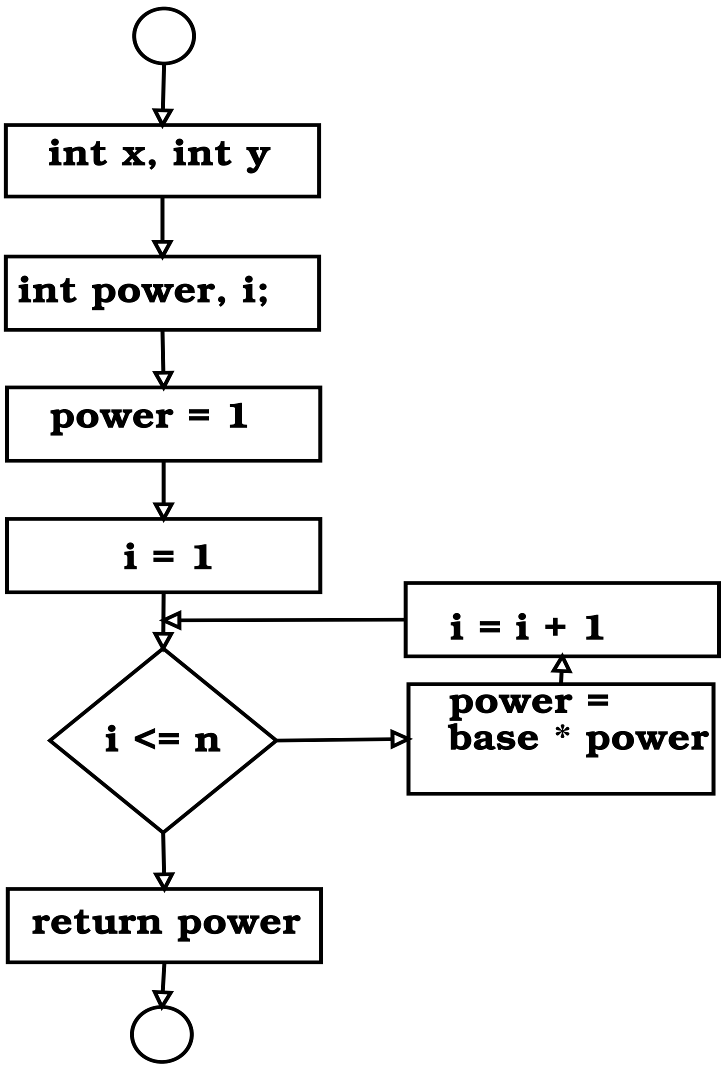
In the sequel we will represent statements by square boxes and branches by diamond boxes.

## Control Flow Graph: Example

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

## Control Flow Graph: Example

The control graph for the power function appears below.



## Control Flow Graphs: Definitions

- An *Execution Path* (or, just *path*) is a sequence of nodes in the control flow graph that starts at the entry node and ends at the exit node.
- A *Branch* (or,*Decision*) is a point in a program where the control flow can diverge. For example, `if-then-else` statements and `switch` statements typically cause branches in the control flow graph.

  In the control flow graph branches are represented by a node node with two or more edges that exiting that node.
- A *Condition* is a simple predicate or simple relational expression occurring within a branch. When the in the conditions in a branch are given certain values, determine which path execution follows in the control flow graph.

  For example, the branch given by

  $$\text{if}\,((\text{a} > 1)\&\&(\text{b} == 0))$$

  consists of the conjunction of two conditions (a ¿ 1) and (b == 0).
- A *Feasible Path* is a path where there is a test case in the input domain that can exercise the path. The converse is an *Infeasible Path* which is a path that is not feasible.

## Static Analysis: Data Flow

The aim of data fbw analysis is to provide information about the creation and use of data definitions in a program.

Certain patterns of data usage can reveal faults in the program. Further, data-fbw analysis can is automatable and provides good indicators of program faults. In data fbw analysis we look for data fbw anomalies:

> Data-fbw anomalies: illogical or useless sequence of data object state changes.

Data fbw anomalies indicate the places where data related errors may be present.

## Data Flow Analysis Concepts

We begin by letting $P$ be the program or component under test and $CFG(P)$ its control fbw graph. We can then determine the following actions on a variable $X$ within the control fbw graph.

**Define (d)** The variable $X$ is defined when it is given (assigned) a value;

**Reference (r)** The variable $X$ is referenced. It can either be referenced as part of a computation (c-reference) or as part of a predicate or condition (p-reference);

**Undefine (u)** The variable $X$ looses its value or its value becomes unknown or uncertain.

## Data Flow Anomalies

The next step is to identify anomalies that can occur when $P$ uses variables. The anomalies are labelled according to the order in which $P$ defines, references or undefines the variables.

**u-r anomaly** indicates that an undefined variable is referenced;

**d-u anomaly** indicates that a defined variable has not been referenced before it becomes undefined.

**d-d anomaly** indicates that the same variable is defined twice causing a hole in the scope of the first defined occurrence of the variable.

**u-u anomaly** indicates that the same variable has been undefined twice.

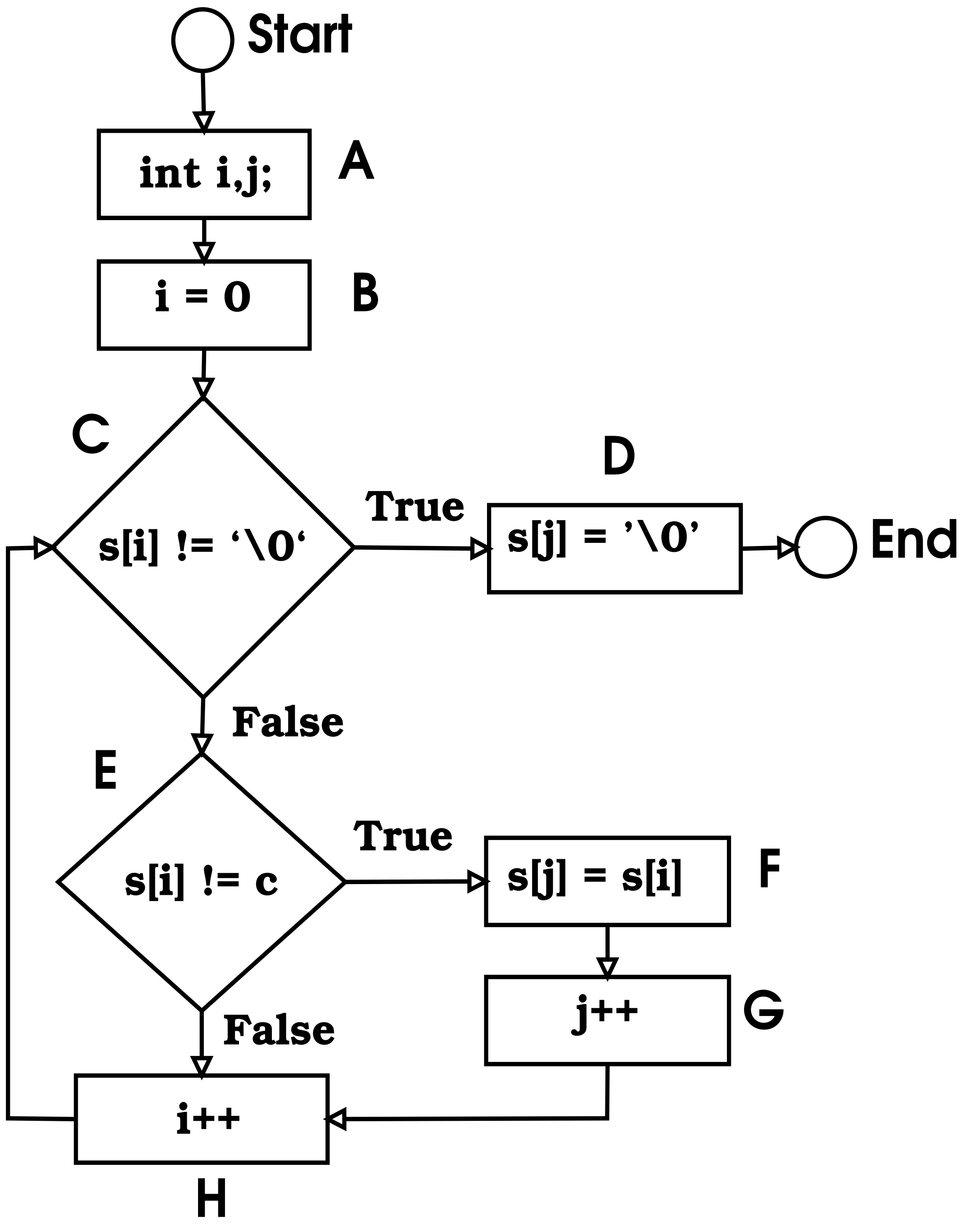
## Data Flow Analysis: Example

Consider the following program for removing c characters from a string s.

```
/* Squeeze: delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;
    for (i = 0; s[i] != ’\0’; i++)
          if (s[i] != c)
               s[j++] = s[i];
   s[j] = ’\0’;
}
```

From *The C Programming Language* by Kernigan and Ritchie

## Data Flow Analysis: Example



## Data Flow Analysis: Example

In this example the component under test is Squeeze and the variables of interest are i, j, s and c.

Examining the path (A, B, C, E, F, G) we see that j has two **u-r** anomalies and another **u-r** anomaly on the path (A, B, C, D).

## Data Flow Analysis: Example

If we change Squeeze as follows:

```
/* Squeeze: delete all c from s */
void squeeze(char s[], int c)
{
    int i, j;
    for (i = j = 0; s[i] != ’\0’; i++)
          if (s[i] != c)
               s[i++] = s[i];
   s[i] = ’\0’;
}
```

we create a number of **d-u** anomalies.

**Exercise 11** *What paths on the control flow graph give rise to the* **d-u** *anomalies.*

## Data Flow Analysis

More generally what types of faults can data fbw analysis detect? Typically we can detect common types of programming errors.

- Typing errors
- Uninitialised variables
- Misspelling of names
- Misplacing of statements
- Incorrect parameters
- Incorrect pointer references

## Data Flow Analysis Approaches

### Static Data Flow Analysis

- Static data fbw analysis is done it without executing the code – it is based purely on the structure of the program;
- It can be a pen and paper approach or done with tools that examine the structure of a program.

### Dynamic Data Flow Analysis

- Instrument the program by inserting additional statements into the program so that information about variables gathered.
- The desired information is obtained by executing the instrumented program for a properly chosen set of input data.

## Static Data Flow Analysis

**Method:**

- Traverse a program’s paths and build up ”path expressions”:
- Path expressions describe the sequence of actions taken on a variable when the program is executed along the path;
- We detect the presence of data fbw anomalies by examining the constituent components of path expressions.

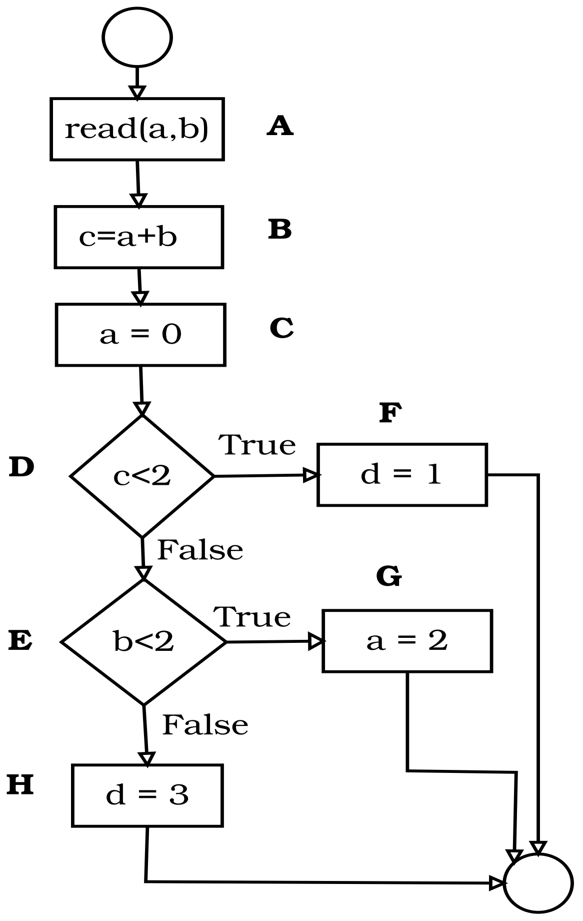
## Static Data Flow Analysis

An example of *Static Data Flow Analysis*.

```
1.    read (a, b);
2.    c = a + b;
3.    a = 0;
4.    if (c < 2)
5.      d = 1;
6.    else
7.    {
8.      if (b < 2)
9.        a = 2;
10.     else
11.       d = 3;
12.   }
```

Consider the variable a on the False-True path. The variable a is defined on line 1, referenced on line 2, defined on line 3, and defined again on line 9: i.e. we have d,r,d,d on this path and a **d-d** anomaly.

## Static Data Flow Analysis



## Dynamic Data Flow Analysis

One method of dynamic data fbw analysis uses *Finite State Automatons (FSA)*. For each program variable we create a FSA with the following states:

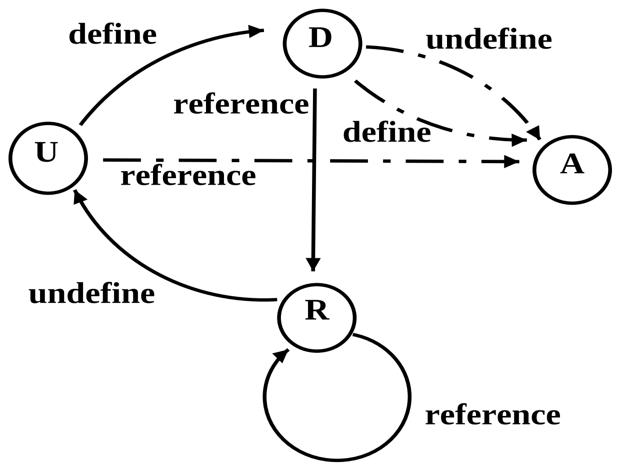
| **State** | **Semantics** |
|---|---|
| U | The variable is undefined; |
| D | The variable is defined; |
| R | The variable is defined and referenced; |
| A | An anomaly has occurred. |

The transitions for the FSA are:

| **Event** | **Semantics** |
|---|---|
| **undefine** | Undefine the variable; |
| **define** | Define the variable; |
| **reference** | Make a reference to the variable; |

# Dynamic Data Flow Analysis

The Finite State Automaton for each variable looks as follows.



- When we enter the **A** state, we stay there. This makes sense since we do not know what condition to reset the variable to if we do not know the exact condition that caused the anomaly. To continue execution we have some options:

  (i) Abort execution after discovering an error to save execution time.

  (ii) Reset to a live state such as **R** after an anomaly and continue.

  (iii) Adopt a more complex state diagram.

- Using the FSA above we do not need to create control paths explicitly. We just need to track the state of each variable in order to infer data flow anomalies.

## Issues in Data Flow Analysis

**Data Flow of Array Elements**

To do data flow analysis for array elements we need to take into account the assignments that can occur between array elements.

- If the array index is a variable or arithmetic expression then we do not know which "*l-value*" we are referencing.
- The calculation of array indexes is a run-time property which complicates static analysis.
- To simplify, static analysis sometimes treats all elements of of an array as if they were a single variable.

  *This is NOT ideal*!

# Data Flow Analysis

## Data Flow Analysis of Array Elements

```
temp = a[j];
a[j] = a[k];
a[k] = temp;
```

The situation above does not cause an anomaly if j!=k, but it does cause and anomaly if j==k. If we treat the array as a single variable, we get a false alarm – a **d-d** anomaly on the final two lines.

## Data Flow Analysis

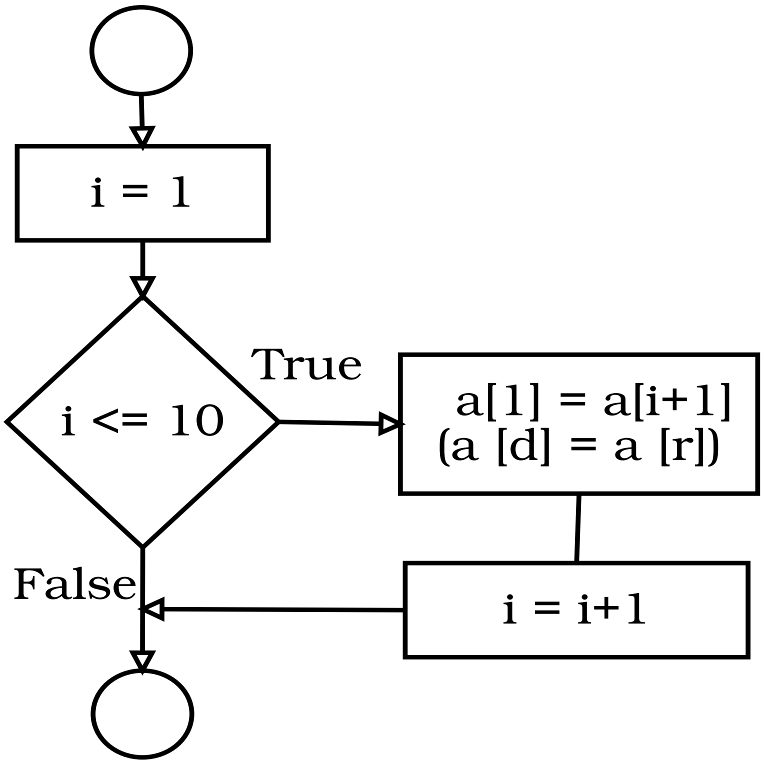
### Data Flow Analysis of Array Elements

```
  i = 1;
  while (i <= 10)
 {
        a[1] = a[i+1];
        // Should be 'a[i] = a[i+1];'
        i = i+1;
  }
```

In this case, by treating the array elements as a single element we will not detect a (*d*-*d*) data fbw anomaly.

So, it is desirable to treat array elements separately. Determining array indices statically on paper is difficult, however, doing it dynamically by program instrumentation is not so hard.

## Static Data Flow Analysis



## Data Flow Analysis

By instrumenting the code we can determine dynamic variables by executing the program:

- The value should already be available if the array index a variable, or it should be easily computable if the array index is an arithmetic expression.
- We need a state variable for each element of an array. A simple structure that can be used is to store the states of elements of an array in the corresponding elements of another array of the same dimension

## Data Flow Analysis

If we were to create an array for the dynamic analysis of

```
a[i,j] = a[i,k] * a[k,j]
```

then we would have:

```
sta[i,k] = f(sta[i,k], r)
sta[k,j] = f(sta[k,j], r)
sta[i,j] = f(sta[i,j], d)
```

## Data Flow Analysis

### Selection of Input Data For Dynamic Analysis

- After instrumenting the program we can detect possible data fbw anomalies by executing the program for a properly chosen set of input data.
- Input data affects the execution paths and therefore the number of anomalies that can be detected by the instrumented program.

## Data Flow Analysis

**Selection of Input Data For Dynamic Analysis**
To detect all the data fbw anomalies that can be detected:

- execute the instrumented program along all possible execution paths;
- execute all loops zero and two times;
  (Based on Huang's Theorem 1.)

## Data Flow Analysis

**Data Flow Analysis for function parameters**?

It is important to track variables when parameters are passed by reference. One approach (Huang) is to use a global queue.

- Before calling a function, put a state variable for each by-reference parameter onto the queue.
- Once inside the function, pull the state variable off the queue.
- Before exiting the function, put the updated state variable onto the queue.
- On return from the function, pull the updated state variable value off the queue.

## Data Flow Analysis

Another approach for function parameters, due to Chan and Chen, is to use an enhanced state diagram.

- Classify the parameters as:
  - (i) Input parameters that are left unaltered during the execution of the subprogram ($P_i$);
  - (ii) Input parameters that also act as output parameters ($P_{io}$);
  - (iii) Input parameters that also act as working variables during the execution of the subprogram ($P_{iw}$);
  - (iv) Output parameters ($P_o$);

## Data-flow Analysis

The calling program causes the following actions of the state machine:

- Treat $P_i$ as undergoing actions **r**;
- Treat $P_{io}$ as undergoing actions **rd**;
- Treat $P_{iw}$ as undergoing actions **ru**;
- Treat $P_o$ as undergoing actions **d**

## Data-flow Analysis

In the called program:

- We give the $P_i$ state variable an initial state of **dr**;
- We give the $P_{io}$ state variable an initial state of **ds**;
- We give the $P_{iw}$ state variable an initial state of **ds**
- We give the $P_o$ state variable an initial state of **u**

Before returning to the calling program we need to

- Check that $P_i$, $P_{io}$ and $P_{iw}$ were referenced;
- Check that $P_{io}$, $P_o$ are not in the **u** state;
- Don't need to worry about $P_{iw}$.

## Data-flow Analysis

Some remarks on Static vs Dynamic data fbw analysis.

(i) Static has difficulties handling arrays and function parameters;
Dynamic handles arrays without too many problems.

(ii) Static approach reveals all data fbw anomalies;
Dynamic approach only detects anomalies along paths that are actually executed.

(iii) Neither is 100Because of this, its advisable to treat static and dynamic as complementary data fbw analysis methods.

## Coverage Based Testing

The aim of coverage based testing methods is to "*cover*" the program or component with test cases.

Put another way, we choose test cases to exercise as much of the program as possible according to some criteria. If some part of the program or component is not exercised then there may well be undiscovered faults lurking there.

## Coverage Based Testing Criteria

- *Statement Coverage* (or, Node Coverage)
  Every statement of the program should be exercised at least once.

- *Branch Coverage* (or, Decision Coverage)
  Every branch (or, decision) of the program should be exercised at least once.

- *Condition Coverage*
  Each condition in a decision takes on all possible outcomes at least once.

- *Multiple-condition coverage*
  All possible combinations of condition outcomes in each decision should be invoked at least once.

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

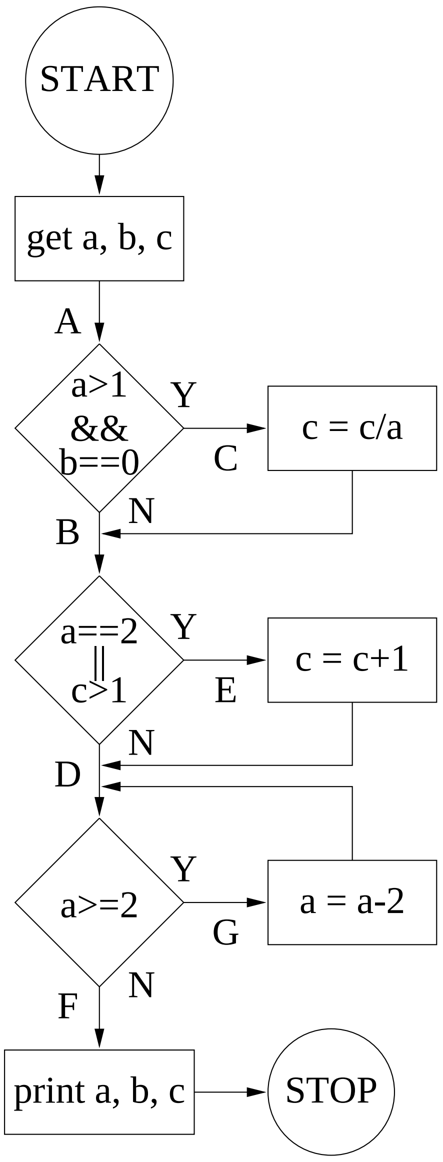
## Coverage Based Testing: Example

Consider the following simple module;

```
void main(void)
{
  int a, b, c;
  scanf("%d %d %d", a, b, c);
  if ((a > 1) && (b == 0))
     c = c / a;
  if ((a == 2) || (c > 1))
     c = c + 1;
  while (a >= 2)
     a = a - 2;
  printf("%d %d %d", a, b, c);
}
```

## Coverage Based Testing

The control fbw graph for the program appears as follows.



Control Flow graph

## Coverage Based Testing

| Coverage criteria | Test cases (a,b,c) | Exec. Paths |
|---|---|---|
| Statement | (2, 0, 3) | ACEGF |
| Branch | (3, 0, 1),<br>(2, 1, 3) | ACDGF,<br>ABEGF |
| Condition | (1, 0, 3),<br>(2, 1, 1) | ABDF,<br>ABEGF |
| Decision/ Condition | (2, 0, 4),<br>(1, 1, 1) | ACEGF,<br>ABDF |
| Multiple Condition | (2, 0, 4),<br>(2, 1, 1),<br>(1, 0, 2),<br>(1, 1, 1) | ACEGF,<br>ABEGF,<br>ABDF,<br>ABDF |
| Path | (2, 0, 4),<br>(2, 1, 1),<br>(1, 0, 2),<br>(4, 0, 0),<br>... | ACEGF,<br>ABEGF,<br>ABDF,<br>ACDGF,<br>... |

# Coverage Based Testing

| Test cases | C1 $a > 1$ | C2 $b==0$ | D1 | C3 $a==2$ | C4 $c > 1$ | D2 | D3 C5 $a \geq 2$ |
|---|---|---|---|---|---|---|---|
| (1,0,3) | F | T | F | F | T | T | F |
| (2,1,1) | T | F | F | T | F | F | T |
| (2,0,4) | T | T | T | T | T | T | T |
| (1,1,1) | F | F | F | F | F | F | F |
| (2,0,4) | T | T | T | T | T | T | T |
| (2,1,1) | T | F | F | T | F | T | T |
| (1,0,2) | F | T | F | F | T | T | F |
| (1,1,1) | F | F | F | F | F | F | F |

D1 is C1 && C2
D2 is C3 | | C4

**Exercise 12** *(i) Can you find an* infeasible path *in this example?*

**Reminder**: *A path is said to be* infeasible *if there are no test cases in the input domain that can execute the path.*

*(ii) How many execution paths are there in the program?*

## Control Flow Coverage Criteria

Recall some of our coverage criteria.

- Statement coverage
- Branch coverage
- Path coverage

Can we also make use of data flow information to guide us in selecting test cases?

## Path Selection Problem

Consider the (schematic) program below.

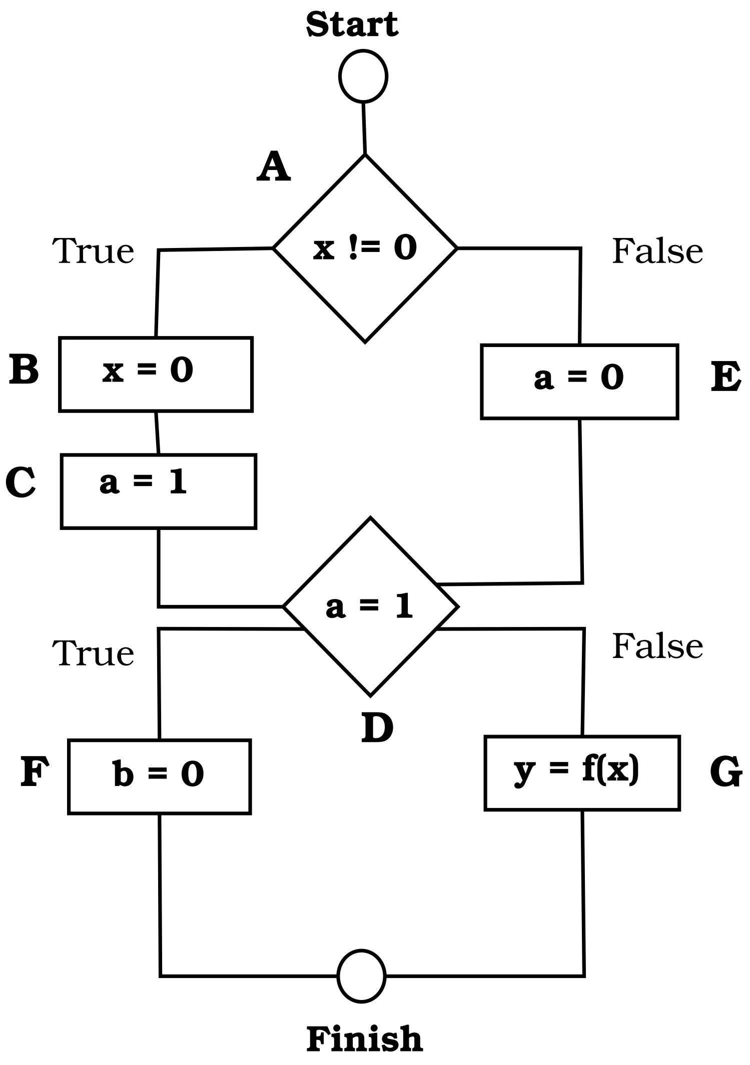
```
    if (x != 0)
    {
       x = 0; a = 1;
    else
       a = 0;
    if (a = 1)
       b = 0
    else
       y = f(x);
```

Can we exercise all paths?

If $\mathtt{x} \neq 0$ then we execute $\mathtt{x} = 0$, $\mathtt{a} = 1$ and $\mathtt{b} = 0$. Otherwise, if $\mathtt{x} = 0$ we execute $\mathtt{a} = 0$ and $\mathtt{y} = \mathtt{f}(\mathtt{x})$.

To execute $\mathtt{x} = 0$ and $y = f(x)$ we would need $\mathtt{x} \neq 0$ and $a \neq 1$ which is impossible to satisfy.

# Path Selection Problem



## Some Coverage Analysis Definitions

**Definition 13**

*(i) Let $d_n(x)$ denote a variable* x *that is assigned a value at node (statement)* n *(***Definition***).*

*(ii) Let $u_m(y)$ denote a variable* y *that is used at node (statement)* m *(***Use***).*

*(iii) A* definition clear path $p$ *with respect to* x *is a sub-path where* x *is not defined at any of the nodes (statements) in $p$.*

*(iv) A definition $d_m(x)$ reaches a use $u_n(x)$ iff there is a sub-path $(m) \bullet p \bullet (n)$ such that $p$ is a definition clear path wrt* x*.*

## Coverage Analysis Criteria

The aim is to define criteria with which to assess test suits. We will only look at a sample of the full set of criteria discussed in the literature but we will examine some of the more popular criteria and give some feel for their relative effectiveness.

In particular we will be looking at *Data Flow Path Selection* criteria due to Rapps and Weyuker.

Frankl, P.G. and Weyuker, E. J., An Applicable Family of Data Flow Testing Criteria, *IEEE Transactions on Software Engineering*, vol 14, no 10 (October 1988), pp 1483-1498.

## Well Formed Data Flow Graphs

First we will assume that we have well formed data fbw graphs.

- There are no edges of the form $(n, n_s)$ or $(n_f, n)$ where $n_s$ is the start node and $n_f$ is the final node.
- No edges of the form $(n, n)$.
- There is at most one edge $(m, n)$ for all $m$ and $n$.
- The CFG is connected.
- There is a single start node and a single final node.
- Every loop has a single entry and a single exit

## Well Formed Data Flow Graphs

We will further assume the following for our CFGs.

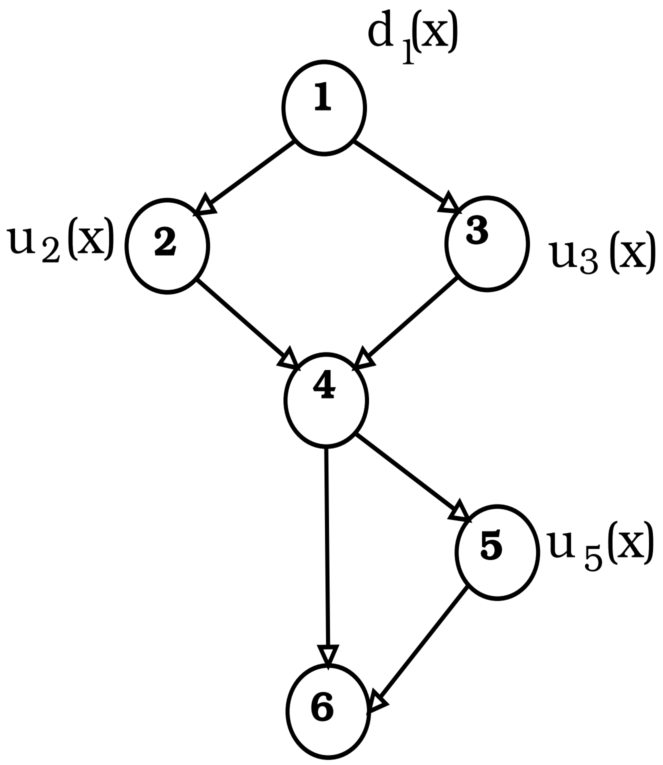
- At least one variable is associated with a node representing a predicate.
- No variable definitions are associated with a node representing a predicate.
- Every definition of a variable reaches at least one use of that variable.
- Every use is reached by at least one definition.
- Every control graph contains at least one variable definition.
- No variable uses or definitions are associated $n_s$ and $n_f$.

## Rapps and Weyuker’s Data Flow Criteria

1. All-Defs
2. All-Uses
3. All-C-Uses, Some-P-Uses
4. All-P-Uses, Some-C-Uses
5. All-P-Uses
6. All-Du-Paths

## Rapps and Weyuker’s Data Flow Criteria All-Defs

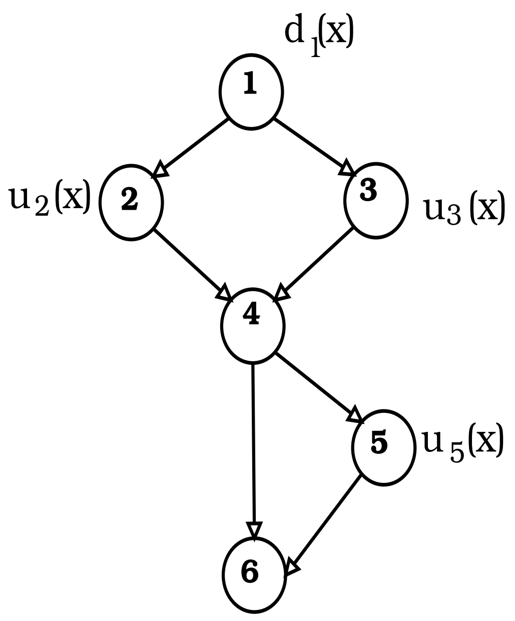
For **All-defs** we require that there is some definition-clear sub-path from each definition to some use reached by that definition.



Test suit requires that we test the paths from definitions to uses. A test case that tests path 1, 2, 4, 6 or 1, 3, 4, 5 are both satisfactory under this criteria.

## Rapps and Weyuker’s Data Flow Criteria All-Uses

The **All-Uses** criteria requires some definition-clear sub-path from each definition to each use reached by that definition and each successor node of the use.



Test suit requires that we test the paths from definitions to uses and their successor nodes. Test cases for the paths 1, 2, 4, 5, 6 and 1, 3, 4, 5, 6 are satisfactory under this criteria. A test case that tests only 1, 2, 4, 5 is not satisfactory.

## Computation Uses and Predicate Uses

**Definition 14**

*(i) A* **C-use** *of a variable is a "*computation use*" of a variable, for example,* $\mathrm{y} = \mathrm{x} * 2$*;*

*(ii) A* **P-use** *of a variable is a "*predicate use*" , for example,* $\mathrm{if}\,(\mathrm{x} < 2) \ldots$*.*

**All-C-Uses**, **Some-P-Uses**

- 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**

- 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.

## Rapps and Weyuker’s Data Flow Criteria All-P-Uses

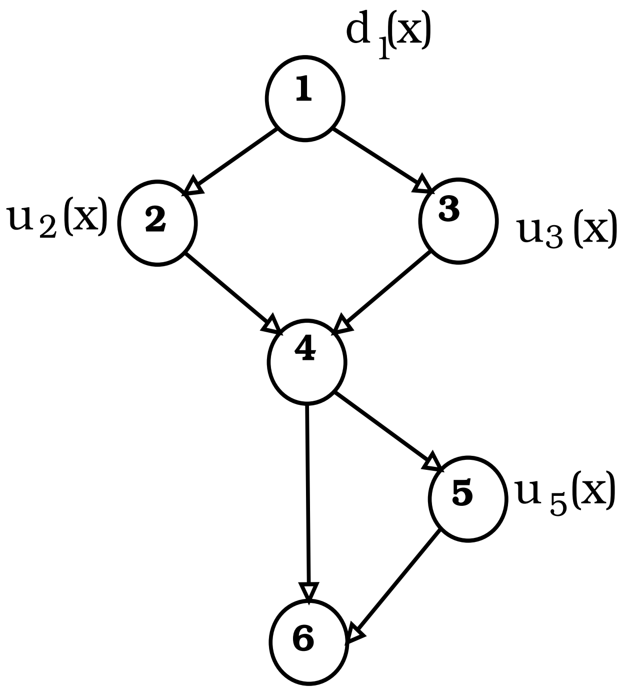
Some definition-clear sub-path from each definition to each P-Use reached by that definition and each successor node of the use

## Rapps and Weyuker’s Data Flow Criteria
## All-DU-Paths

Here **DU** stands for definition-use.

All definition-clear sub-paths that are cycle-free or simple-cycles from each definition to each use reached by that definition and each successor node of the use.

## All-DU-Paths Example



The DU criteria requires that we test $d_1(x)$ to each use.

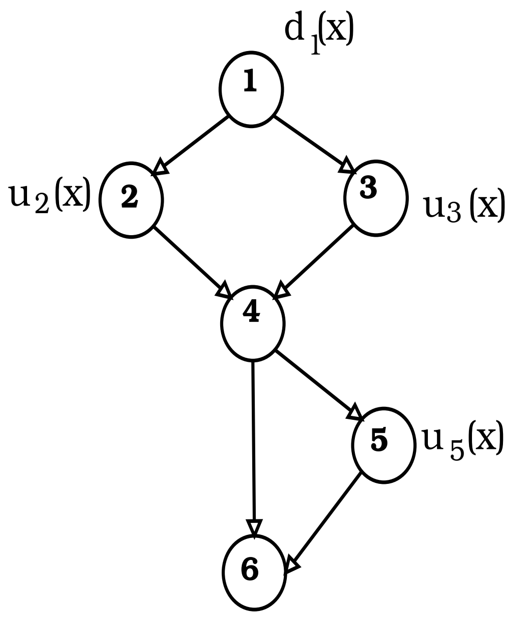
(i) $d_1(x)$ to a $u_2(x)$;

(ii) $d_1(x)$ to a $u_3(x)$;

(iii) both paths from $d_1(x)$ to $u_5(x)$.

Under this criteria, satisfactory paths are given by 1, 2, 4, 5, 6 and 1, 3, 4, 5, 6.

## Rapps and Weyuker's Data Flow Criteria All-Uses



The All-Uses criteria requires that we test $d_1(x)$ to each use and its successor nodes.

(i) $d_1(x)$ to a $u_2(x)$;

(ii) $d_1(x)$ to a $u_3(x)$; and

(iii) $d_1(x)$ to $u_5(x)$.

Under this criteria, satisfactory paths are given by 1, 2, 4, 5, 6 and 1, 3, 4, 6.

## Comparing Coverage Criteria

The way to compare these criteria is to defined a specific relation between the criteria. The relation in question is called *subsumption* and is defined as follows.

**Definition 15** Criterion $A$ subsumes criterion $B$ iff for any control flow graph $P$:

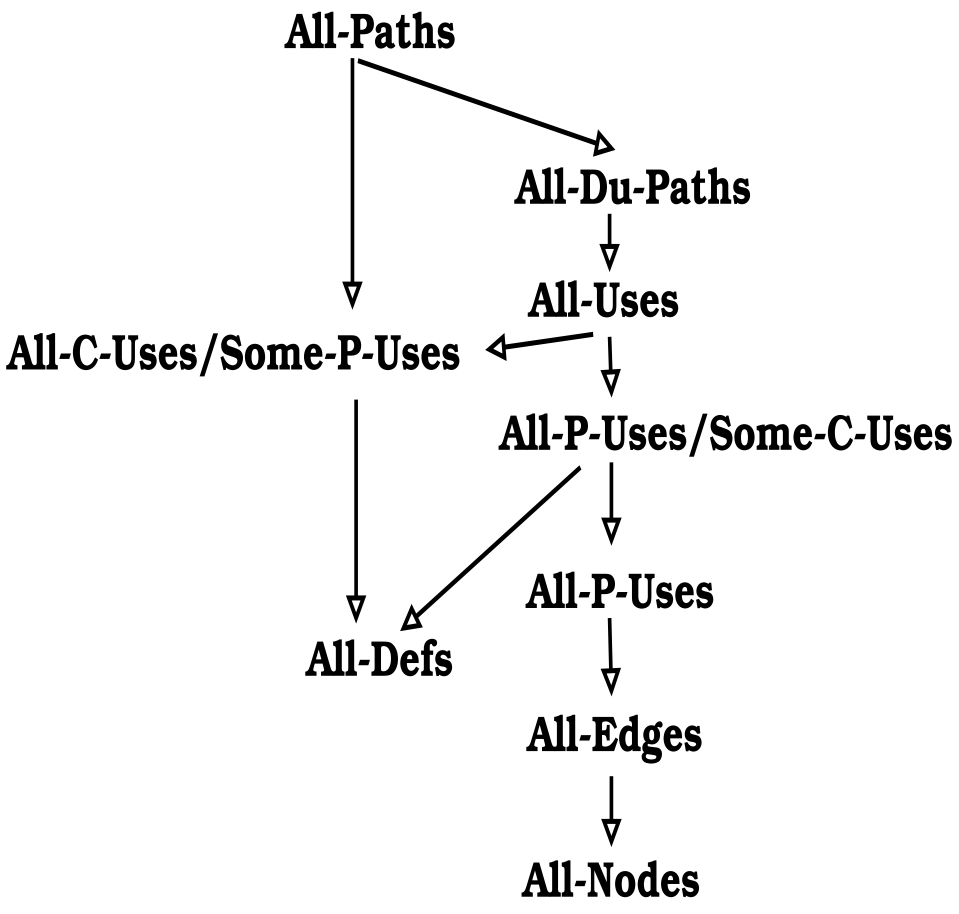
$$P\,\mathrm{satisfies}\,A \Rightarrow P\,\mathrm{satisfies}\,B.$$

Criteria A is equivalent to criteria B iff A subsumes B, and B subsumes A.

*How can we compare these criteria*?

All criteria that we have studied select a set of paths, so we compare the paths that each criteria selects. Note, however, that the set of paths that satisfy a criterion are not necessarily unique.

## Comparing Coverage Criteria
## A Lattice of Coverage Criteria



## Conclusions

- An improvement over pure control fbw techniques;
- Provides a rationale for which combinations of sub-paths to consider;
- Most commonly used criteria is all-uses;
- One problem with data fbw coverage is infeasible paths
  We don’t usually get 100% coverage.

# Object-Oriented Testing

## Testing Object Oriented Programs

Object oriented systems provide many advantages to the system designer and system programmers, but in these notes we explore their testability.

- Do Object Oriented systems make testing harder or easier?
- Does good code reuse lead to test case reuse?
- Do the techniques that we have explored earlier in this course suffice for testing Object Oriented programs or do we need to change existing techniques?
- Do we need to develop new techniques?

## Object-Oriented Programming Languages

Object Oriented programming languages support a number of features that are aimed at making the design, maintenance and reuse of code much easier.

OO programming languages support abstract data types (ADTs).

- Abstract data types provide *Information hiding*, that is, only the details relevant to the problem or the services provided by an object are visible. Irrelevant detail is hidden. C++ and Java both provide different storage classes such as private and public operations for information hiding.
- Abstract data types *Encapsulate* data and operations, that is, they package together data and the operations that act on that data.

## Object-Oriented Programming Languages

- OO programming languages support inheritance as a convenient structuring mechanism for programmers.
  - Changes to a parent type are reflected in the children; and
  - OO programming languages support dynamic binding, or polymorphism.
- OO programming languages supports reuse.

## Object Oriented Programs

Each class defines a type where the instances (or objects) of the class are to be thought of as the members of the type.

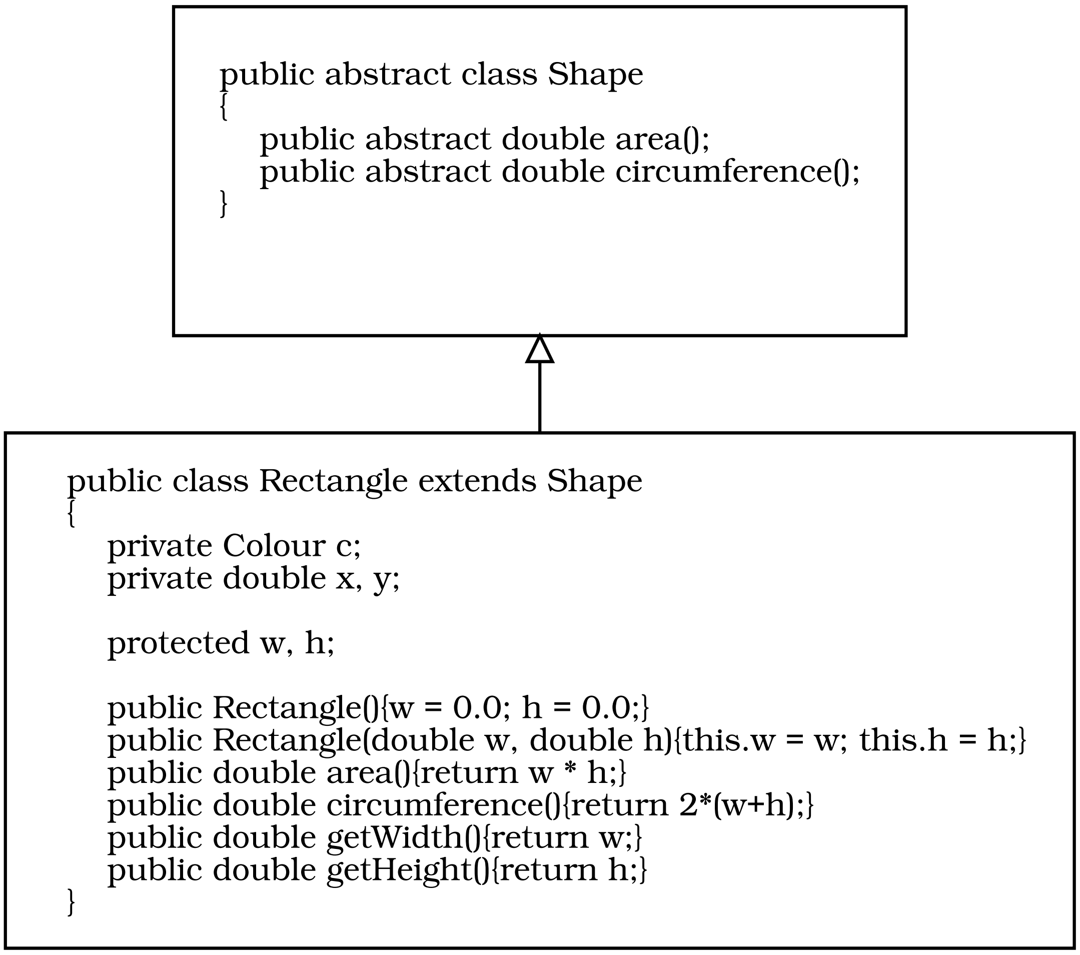
Every member of a class has:

- Access methods;
- Instance variables (attributes).

  Any access method may have access to the instance variables.
- An object is an instance, or an element, of a class.
  - There may be multiple instances of a class, each with its own instance variables.
  - Methods are typically invoked via message passing.
- Object oriented programs use "*Dynamic Binding*".

# Object Oriented Programs

**An abstract class**



```
public abstract class Shape
{
    public abstract double area();
    public abstract double circumference();
}
```

```
public class Rectangle extends Shape
{
    private Colour c;
    private double x, y;

    protected w, h;

    public Rectangle(){w = 0.0; h = 0.0;}
    public Rectangle(double w, double h){this.w = w; this.h = h;}
    public double area(){return w * h;}
    public double circumference(){return 2*(w+h);}
    public double getWidth(){return w;}
    public double getHeight(){return h;}
}
```

**Rectangle inherits Shape and overrides its methods to implement them.**

## Object Oriented Programs

In the case of *Single Inheritance* a class may only inherit from from one, and only one, parent class.

In the case of *Multiple Inheritance* a class may inherit from one or more parent classes.

The parent class is called the *Superclass* and the child is called the *Subclass*.

## Issues in Object Oriented Testing

- What is the basic unit for unit testing?
- What are the implications of encapsulation?
- What are the implications of inheritance?
- What are the implications of polymorphism/dynamic binding?
- What are the implications for testing techniques?
- What are the implications for testing processes?

## The Basic Unit for OO Unit Testing

In procedural programming the basic component for unit testing is typically a subroutine (functional or procedure call).

- The testing method is (typically) input/output based where we need to execute subroutines in the process of testing.
- In functional testing strategies, once a subroutine is tested and we have confidence in it, it is usually not re-tested unless *the subroutine itself* is changed.

## The Basic Unit for OO Unit Testing

In object-oriented programming the basic component for testing is:

$$\text{class} = \text{data structure} + \text{set of operations}.$$

Objects are instances of classes where the internal states of the object are relevant to the testing of an object. Thus, the correctness of an object is not based only on the output given by method calls, but also on the internal state of the object.

## The Basic Unit for OO Unit Testing

Classes are the natural unit for unit test case design.

- Methods are meaningless apart from their class. For example, a method may require an object to be in a specific state before it can be executed, where that state can only be set by another method (or combination of methods) in the class.

## Implications for Encapsulation

Encapsulation is not usually a source of errors (more often its the converse) but it may be an obstacle to testing.

- How can we get the concrete state of an object? We need to break the encapsulation to perform the test by:
  - Using features of the language (e.g. C++ friend); or
  - Using low level probes or debugging tools to manually inspect the state.

## Implications for Encapsulation

How can we get the concrete state of an object?

- We can use the abstraction by constructing scenarios based on sequences of events.
  - Execute the methods in the classes to examine sequence of events;
  - The object *State* is implicitly inspected via the access methods.
- Use or provide “*hidden*” functions to examine the state (Instrumenting the code). Instrumenting the code in such a way is extremely useful for debugging the system throughout the life of the system.

## Implications of Inheritance

Inherited features often require re-testing – a new context for the use of the class results when features are inherited.

- Multiple inheritance increases the number of contexts to test.

Specialisation relationships should correspond to problem domain specialisation. The re-usability of superclass test cases depends on this idea.

## Implications of Inheritance

Which functions must be tested in a subclass?

```
class parent {
      void foo(int xx);
      intrange(); // returns between 1-10
}

class child extends parent {
      intrange(); // returns between 1-20
}
```

- When testing the child, we need to retest `range()` because of the overloading in the subclass.
- Do we need to retest `foo()`? Suppose `foo()` contains the line

  $$\mathtt{x} = \mathtt{x}/\mathtt{intrange}();$$

  In this case `foo()` depends on `intrange()` and re-testing is necessary, but maybe we do not need to retest completely.

## Implications of Inheritance

*Can tests for a parent class be reused for a child class*?

- Observe that `parent.range()` and `child.range()` are two different functions with different specifications and implementations.
- Test cases are derived from the different specifications as well as the implementation, however, the functions are likely to be similar, so the provided we use principles such as the "*open/closed principle*" in design, the greater the overlap in testing.
- New tests are those for `child.range()` requirements that are not satisfied by the `parent.range()` tests.

## Implications of Inheritance

Consider the following program fragment.

```
Parent::describeSelf()
{
  if (val < 0) message("Less");
  else if (val == 0) message ("Equal");
  else message("More");
}

Child::describeSelf()
{
  if (val < 0) message("Less");
  else if (val == 0) message ("Zero Equal");
  else
  {
     message("More");
     if (val == 42) message("Jackpot");
  }
}
```

The tests for the parent and child classes appear in the following table.

| **Value** | **Parent Response** | **Child Response** | **Test Changes** |
|---|---|---|---|
| -1 | Less | Less | OK |
| 0 | Equal | Zero Equal | *Changed* |
| 1 | More | More | OK |
| 42 | | Jackpot | *Add* |

## Implications of Inheritance

One approach to inheritance testing is to *Flattening the Inheritance Structure*.

- Each subclass is tested as if all inherited features were newly defined.
- Tests used in super-classes can be reused but many tests will be redundant.

## Implications of Inheritance

Another approach is *Incremental Inheritance-based Testing*

- First test each base class by: (1) Testing each method; and (2) Testing interactions among methods.
- Then, consider all sub-classes that use only previously tested classes.
- 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.

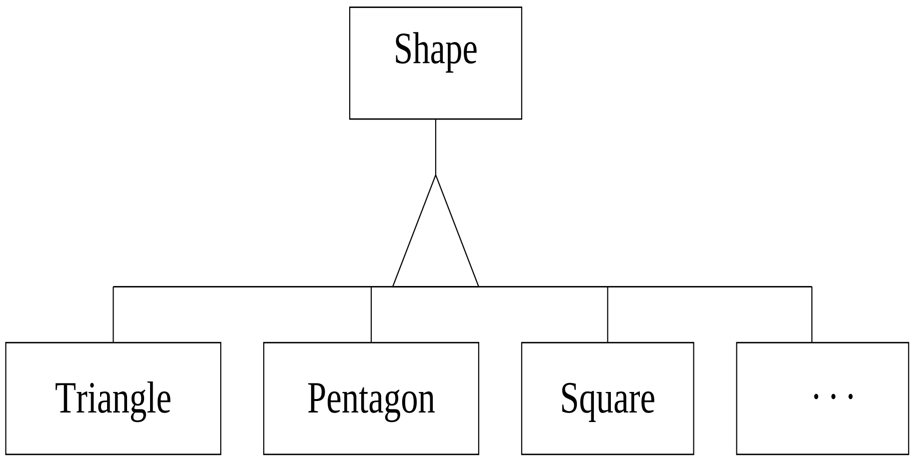
## Implications of Inheritance

Incremental inheritance-based testing:

- Saves time by reducing the number of test cases;
- Reduces the execution time since fewer test cases are needed than for a flattened hierarchy;
- Reduces the number of test results that need to be evaluated, but it may increase the cost of selecting new test cases, especially in determining what has changed and what is new.
- Inheritance based testing is a form of regression testing where the aim is to minimise the number of test cases needed to exercise a modified class.

## Implications of Polymorphism

Consider the following inheritance hierarchy.



```
void resize(Shape polygon)
{
    ...
    data = polygon.area();
    ...
}
```

The implementation of area that actually gets called may depend on the state and the runtime environment.

## Implications of Polymorphism

In procedural programming, procedure calls are statically bound.

In the case of object oriented programming each possible binding of a polymorphic component requires a separate set of test cases.

- However, it may be hard to find all such bindings – after-all the exact binding used in a particular instance may only be known at run-time.
- Dynamic binding also complicates integration planning. Many server classes may need to be integrated before a client class can be tested.

## Implications of Polymorphism

One approach to the dynamic binding problem is to reduce combinatorial explosion in the number of test cases that cover all possible combinations of polymorphic calls:

- Use analysis (e.g. program domain testing or program slicing) to determine possible bindings
- *Note:* In most systems the average number of ”possible” bindings is 2

## Implications for Testing Techniques

Here we need to consider:

- Black-box testing
- White-box testing
- State-based testing

## Black-box Testing

*Conventional black-box methods are useful for object-oriented systems!*

We don’t need the details of the internal states for objects.

## White-box Testing

White-box techniques can be adapted to method testing, but are not sufficient for testing classes (why?).

Part of the answer is that methods can influence each other through the object state. In this case the coverage or domain criteria can become dependent on the internal state.

In turn the state of an object may well depend on the preceding sequence of object method calls and their parameter values.

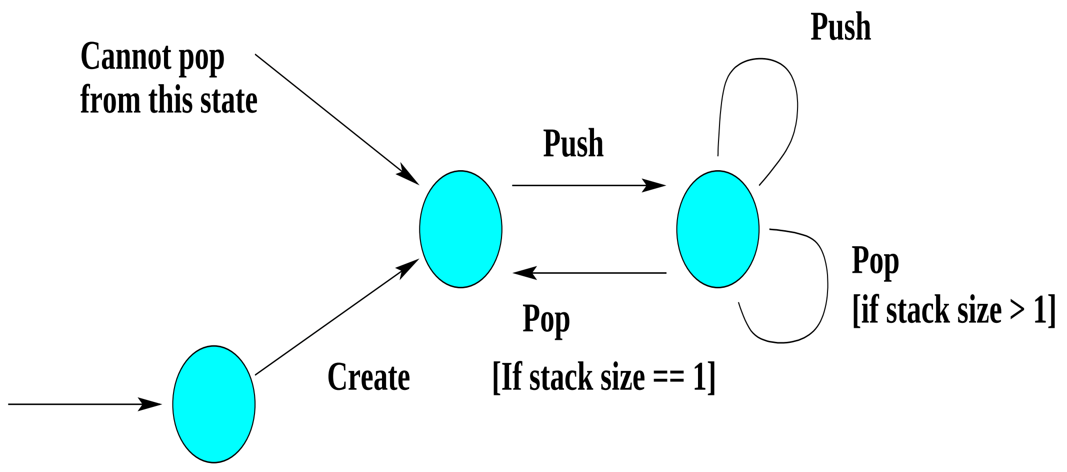
## State-based Testing

In the case of state-based testing we can derive test cases by modelling a class as a state machine.

- Begin by identifying the stating, exiting and legal states for the class;
- Methods cause state transitions;
- The state model now defines allowable sequences of method calls, for example, can't pop from a stack until we push something on to it;
- Test cases are then devised to exercise each transition.

## State-based Testing

A simple state model of a stack.



## State-based Testing

State models are often created as part of a design methodology.

For example, UML uses state-charts precisely for the purpose of specifying and understanding the legal sequences of actions on an object.

## State-based Testing

Of course there are some problems with state based testing!

- It may take a very lengthy sequence of operations to get an object into some desired state.
- State based testing may not be useful if the class is designed to accept any possible sequence of method calls.
- State control may be distributed over an entire application with methods from other classes referencing the state of the class under test.

  System-wide control makes it difficult to verify a class in isolation
- We often need a global state model to show how classes interact

## Implications for Testing Processes

*Unit Testing*

- Unit testing focuses on behaviour of individual units (re-call that classes best serve as units in OO testing).
- Tests are derived from module specifi cations or source code.
- Drivers and stubs usually required

*Integration Testing*

- Integration testing focuses on communication and inter-face problems.
- Test cases are derived from module interfaces and de-tailed architecture specifi cations
- Some drivers and stubs may be required.

*Regression Testing*

- Re-testing fi xed/modifi ed code

*System testing*

- System testing focuses on behaviour of the system as a whole.
- Test are derived from requirements specifi cations.
- Code seen as a black box
- Drivers and stubs usually not needed

## Unit Testing

The OO context changes what we normally understand by a *unit*.

The changes concern:

- The state of instance variables; and
- Sequences of method calls.

We need to test a class and its subclasses.

## Integration Testing

There is a strong need to test component interaction, but to do this we need to understand how objects interact!

There is also a need to test specific contexts such as dynamically bound variables and parameters, and polymorphic operators.

Object-oriented testing strategies include:

- Thread-based strategies; and
- Uses-based strategies.

## Integration Testing

Object-oriented testing strategies:

**Thread-based**
- A thread consists of all the classes needed to respond to a set of related external inputs or events;
- Each class is unit tested, and then the set of classes in the thread is exercised.

**Uses-based**
- Begin by testing classes that use few or no other classes;
- Then, test classes that use the first group of classes;
- Follow this by testing the classes that use the second group, and so on;
- Create stubs/drivers as necessary.

## Regression Testing

Changes may have greater impact because of inheritance problems discussed earlier.

## System Testing

System testing is not usually impacted!

## Summary

- *Abstract data types*
  - Well-defined interfaces and centralised focus can help with testing
- *Inheritance* Increases the reuse of classes, and thus reuse of test cases, but
  - the impact of changes must be carefully assessed and taken into account
- *Dynamic binding*
  - Simplifies code, but testing must consider all ”possible” bindings;
  - Beware of ”hidden” interactions!