Brief Course Overview

- Introduction to Measurement Theory
- Measurement as applied Software
- Examples of Various Metrics, Measures and Indicators
What is measurement?

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the world according to clearly defined rules.
The importance of Measurement

Can software be measured?

Is it software measurement useful?

How do you measure software?
The importance of Measurement

- Measurement is **crucial** to the progress of all sciences, *even Computer Science*
- Scientific progress is made through
  - Observations and generalisations…
  - …based on data and measurements
  - Derivation of theories and…
  - …confirmation or refutation of these theories
- Measurement turns an art into a science
Uses of Measurement

- Measurement helps us to **understand**
  - Makes the current activity visible
  - Measures establish guidelines
- Measurement allows us to **control**
  - Predict outcomes and change processes
- Measurement encourages us to **improve**
  - When we hold our product up to a measuring stick, we can establish quality targets and aim to improve
Some propositions

- Developers who drink coffee in the morning produce better code than those who do drink orange juice.
- The more you test the system, the more reliable it will be in the field.
- If you add more people to a project, it will be completed faster.
Abstraction Hierarchy

Abstract World

- Theory
- Proposition (An assumption or set of assumptions which are taken to be true.)

Empirical World

- Hypothesis (An proposed explanation of a phenomenon.)
  - Data Analysis
    - Take measurements, carry out studies, look at past data, etc.
    - Proof or refute the hypothesis.
Adding more developers to a late project will only make it later.

If a project misses one or more milestone deadlines, it is considered to be late.

The greater the difference between the current time and the missed milestone, the later the project is said to be.

Project p has n developers working on it
Project p missed a deadline be x days (x > 0)
If we add m developers (m > 0), then when the milestone is reached, the project would be y days late (y > x)

Carry out studies, analyse data from past projects,… Verify or disprove the theory
Definitions (1/2)

- **Theory** - A supposition which is supported by experience, observations and empirical data.

- **Proposition** – A claim or series of claims which are assumed to be true.

- **Hypothosis** – A proposed explanation for a phenomenon. Must be *testable* and based on previous observations or scientific principles.
**Definitions** (2/2)

- **Entities** – Objects in the real world. May be animate, inanimate or even events.

- **Attributes** – Characteristics / features / properties of an entity

**Example**

*Entity:* Program  
*Attributes*  
- Time to Develop  
- Lines of code  
- Number of Defects
Levels of Measurement

Various scales of measurements exist:

- Nominal Scale
- Ordinal Scale
- Interval Scale
- Ratio Scale
The **Nominal Scale**

Example: *A religion nominal scale*

<table>
<thead>
<tr>
<th>Joe</th>
<th>Michelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rachel</td>
<td>Christine</td>
</tr>
<tr>
<td>Michael</td>
<td>James</td>
</tr>
<tr>
<td>Clyde</td>
<td>Wendy</td>
</tr>
</tbody>
</table>

- Catholic
- Muslim
- Other
- Jewish
The **Nominal Scale** (2/2)

- The most simple measurement scale
- Involves sorting elements into categories with regards to a certain attribute
- There is no form of ranking
- Categories must be:
  - Jointly exhaustive
  - Mutually exclusive
The Ordinal Scale

Example: A *degree-classification ordinal scale*

<table>
<thead>
<tr>
<th>1st Class</th>
<th>2nd Class</th>
<th>3rd Class</th>
<th>Failed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Joe</td>
<td>Michelle</td>
<td>Michael</td>
<td>Clyde</td>
</tr>
<tr>
<td>Rachel</td>
<td>Christine</td>
<td>James</td>
<td>Wendy</td>
</tr>
</tbody>
</table>
The Ordinal Scale (2/2)

- Elements classified into categories
- Categories are ranked
- Categories are transitive $A > B \land B > C \Rightarrow A > C$
- Elements in one category can be said to be better (or worse) than elements in another category
- Elements in the same category are not rankable in any way
- As with nominal scale, categories must be:
  - Jointly exhaustive
  - Mutually exclusive
Interval Scale

- Indicates exact differences between measurement points
- Addition and subtraction can be applied
- Multiplication and Division **CANNOT** be applied
- We can say that product D has 8 more crashes per month but we cannot say that it has 3 times as more crashes

<table>
<thead>
<tr>
<th>Temperature of Different CPUs</th>
<th>CPU A</th>
<th>CPU B</th>
<th>CPU C</th>
<th>Product D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0°C</td>
<td>30°C</td>
<td>60°C</td>
<td>120°C</td>
<td></td>
</tr>
<tr>
<td>86°F</td>
<td>140°F</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Ratio Scale

- The highest level of measurement available
- When an absolute zero point can be located on an interval scale, it becomes a ratio scale
- Multiplication and division can be applied (product D crashes 4 times as much per month than product B)
- For all practical purposes almost all interval measurement scales are also ratio scales
Scales are hierarchical
Each higher-level scale possesses all the properties of the lower ones
A higher-level of measurement can be reduced to a lower one but not vice-versa
Measures, Metrics and Indicators

- **Measure** – An appraisal or ascertainment by comparing to a standard. E.g. Joe’s body temperature is 99° Fahrenheit

- **Metric** – A quantitative measure of the degree to which an element (e.g. software system) given attribute.
  - E.g. 2 errors were discovered by customers in 18 months (more meaningful than saying that 2 errors were found)

- **Indicator** – A device, variable or metric can indicate whether a particular state or goal has been achieved. Usually used to draw someone’s attention to something.
  - E.g. A half-mast flag indicates that someone has died
Example of a Measure
Example of a Metric

![Graph showing the relationship between bugs and months since release. The graph has a red line that increases as the x-axis (months since release) increases. The y-axis represents the number of bugs.]
Example of a Indicator

Indicator of maximum safe temperature

Time (Hours)

Temp.
Some basic measures (1/2)

- **Ratio**
  - E.g. The ratio of testers to developers in our company is 1:5

- **Proportion**
  - Similar to ratio but the numerator is part of the denominator as well
  - E.g. \[
    \frac{\text{Number of satisfied customers}}{\text{Total number of customers}}
  \]
Some basic measures (2/2)

- **Percentage**
  - A proportion or ration express in terms of per hundred units
  - E.g. 75% of our customers are satisfied with our product

- **Rate**
  - Ratios, proportions and percentages are static measures
  - Rate provides a dynamic view of a system
  - Rate shows how one variable changes in relation to another (one of the variables is usually time)
  - E.g. Lines of Code per day, Bugs per Month, etc
Reliability and Validity of Measurements

- **Reliability** – Refers to the consistency of a number of measurements taken using the same measurement method.

- **Validity** – Refers to whether the measurement or metric really measures what we intend it to measure.
Reliability and Validity of Measurements

- Reliable but not valid
- Valid but not reliable
- Reliable and Valid
Measuring Software
What makes quality software?

- Cheap?
- Reliable?
- Testable?
- Secure?
- Maintainable?
- …
What makes quality software?

- There is not a clear-cut answer.
- It depends on:
  - Stakeholders
  - Type of system
  - Type of users
  - ...
- Quality is a multifaceted concept.

Different ideas about a quality car.
Different Quality Scenarios

- Online banking system
  - Security
  - Correctness
  - Reliability

- Air Traffic Control System
  - Robustness
  - Real Time Responses

- Educational Game for Children
  - Userfriendliness
The 3 Ps of Software Measurement

With regards to software, we can measure:

- **Product**
- **Process**
- **People**
Measuring the Product

- Product refers to the actual software system, documentation and other deliverables
- We examine the product and measure a number of aspects:
  - Size
  - Functionality offered
  - Cost
  - Various Quality Attributes
Measuring the Process

- Involves analysis of the way a product is developed
- What lifecycle do we use?
- What deliverables are produced?
- How are they analysed?
- How can the process help to produce products faster?
- How can the process help to produce better products?
Measuring the People

- Involves analysis of the people developing a product
- How fast do they work?
- How much bugs do they produce?
- How many sick-days do they take?
- Very controversial. People do not like being turned into numbers.
The Measuring Process

Measurement Programme

Non-intrusive Data Collection

Products

Processes

People

Results, Trends, Reports, etc

Modifications
Collecting Software Engineering Data

- **Challenge:** Make sure that collected data can **provide useful information** for project, process and quality management **without being a burden** on the development team.
- Try to be as unintrusive as possible
- Try to make data collection automatic
- Can expensive
  - Sometimes difficult to convince management
A possible collection methodology:

1. Establish the **goal** of data collection
2. Develop a **list of questions** of interest
3. Establish data **categories**
4. Design and test data **collection forms/programs**
5. Collect and validate **data**
6. **Analyse** data
Examples of Metrics Programmes

Motorola

- 7 Goals
  - Improve Project Planning
  - Increase defect containment
  - Increase software reliability
  - Decrease defect density
  - Improve customer service
  - Reduce the cost of non-conformance
  - Increase software productivity

- Various Measurement Areas
  - Delivered defects, process effectiveness, software reliability, adherence to schedule, time that problems remain open, and more…
Examples of Metrics Programmes

**IBM**

- IBM have a Software Measurement Council
- A set of metrics called 5-Up are defined and deal with:
  - Customer Satisfaction
  - Postrelease Defect Rates
  - Customer problem calls
  - Fix response time
  - Number of defective fixes
Examples of Metrics Programmes

Hewlett-Packard

- Heavily influenced by defect metrics
  - Average fixed defects/working day
  - Average engineering hours / fixed defect
  - Average reported defects/working day
  - Defects / testing time
  -...

Product Metrics
What can we measure about a product?

- Size metrics
- Defects-based metrics
- Cost-metrics
- Time metrics
- Quality Attribute metrics
Size Metrics

- Knowing the size of a system was important for comparing different systems together.
- Software measured in lines of code (LOC).
- As systems grew larger KLOC (thousands of lines of code) was also used.
The problems with LOC

- Same system developed with different programming languages will give different LOC readings.
The problems with LOC

- Same system developed by different developers using the same language will give different LOC readings

Video Rental System

- Developer A: 2 KLOC
- Developer B: 1.2 KLOC
- Developer C: 2.5 KLOC
The problems with LOC (3/3)

- To calculate LOC you have to wait until the system is implemented.
- This is not adequate when management requires prediction of cost and effort.
- A different approach is sometimes necessary...
Function Points

- Instead of measuring size, function points measure the *functionality* offered by a system.
- Invented by Albrecht at IBM in 1979
- Still use today: [http://www.ifpug.org](http://www.ifpug.org)
Overview of Function Points

- Function Points gauge the functionality offered by a system.
- A **Function** can be defined as a collection of executable statements that performs a certain task.
- Function points can be calculated before a system is developed.
- They are language and developer independent.
A function point count is calculated as a weighted total of five major components that comprise an application...

- External Inputs
- External Outputs (e.g. reports)
- Logical Internal Files
- External Interface Files – files accessed by the application but not maintained by it
- External Inquiries – types of online inquiries supported
The *simplest* way to calculate a function point count is calculated as follows:

\[(\text{No. of external inputs } \times 4) + (\text{No. of external outputs } \times 5) + (\text{No. of logical internal files } \times 10) + (\text{No. of external interface files } \times 7) + (\text{No. of external enquiries } \times 4)\]
Consider the following system specs:

Develop a system which allows customers to report bugs in a product. These reports will be stored in a file and developers will receive a daily report with new bugs which they need to solve. Customers will also receive a daily status report for bugs which they submitted. Management can query the system for a summary info of particular months.

- 1 External Inputs
- 1 Logical Internal Files
- 2 External Outputs
- 1 External Enquiries
Function Points Example

External Inputs: 1
External Outputs: 2
Logical Internal Files: 1
External Interface Files: 0
External Enquiries: 1
Total Functionality is \((1 \times 4) + (2 \times 5) + (1 \times 10) + (0 \times 7) + (1 \times 4) = 28\)
Function Point Extensions

- The original function points were sufficient but various people extended them to make them more expressive for particular domains.

- Examples
  - General System Characteristics (GSC) Extension
  - 3D Function Points for real time systems
  - Object Points
  - Feature Points
The **GSC** Function Points Extension  

- **Reasoning**: Original Function Points do not address certain functionality which systems can offer  
  - E.g. Distributed functionality, performance optimisation, etc  
- The GSC extension involves answering 14 questions about the system and modifying the original function point count accordingly
1. Data communications  
2. Distributed Functions  
3. Performance  
4. Heavily used configuration  
5. Transaction rate  
6. Online Data Entry  
7. End-user Efficiency  
8. On-line update  
9. Complex Processing  
10. Reusability  
11. Installation ease  
12. Operational Ease  
13. Multiple sites  
14. Facilitation of Change
The analyst/software engineer assigns a value between 0 and 5 to each question. 
- \( 0 = \text{not applicable} \) and \( 5 = \text{essential} \)

The Value-Adjustment Factor (VAF) is then calculated as:

\[
VAF = 0.65 + 0.01 \sum_{i=1}^{14} Ci
\]

You then adjust the original function point count as follows:

\[
FP = FC \times VAF
\]
Consider the bug-reporting system for which we already looked at and suppose the analyst involved answers the GSC questions as follows…

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data communications</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Distributed Functions</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Performance</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>Heavily used configuration</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Transaction rate</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Online Data Entry</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>End-user Efficiency</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>On-line update</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>Complex Processing</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>Reusability</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>Installation ease</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>Operational Ease</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Multiple sites</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>Facilitation of Change</td>
<td>0</td>
</tr>
</tbody>
</table>

Total GSC Score = 25
As you may remember, when we calculated the function point count for this system, we got a result of 28.

If we apply the GSC extension, this count will be modified as follows.

\[
\text{VAF} = 0.65 + (0.01 \times 25) = 0.9 \\
\text{FC} = 28 \times 0.9 = 25.2
\]

Note that the GSC extension can increase or decrease the original count.

In larger systems, the GSC extension will have a much more significant influence on the Function Point Count.
Defect Density

- A metric which describes how many defects occur for each size/functionality unit of a system
- Can be based on LOC or Function Points

\[
\frac{\# defects}{system\_size}
\]
Failure Rate

- Rate of defects over time
- May be represented by the $\lambda$ (lambda) symbol

$$\lambda = \frac{R(t_1) - R(t_2)}{(t_2 - t_1) \times R(t_1)}$$

where,

- $t_1$ and $t_2$ are the beginning and ending of a specified interval of time
- $R(t)$ is the reliability function, i.e. probability of no failure before time $t$
Calculate the failure rate of system X based on a time interval of 60 days. The probability of no failure at time day 0 was calculated to be 0.85 and the probability of no failure on day 5 was calculated to be 0.20.
Example of Failure Rate

\[ \lambda = \frac{R(t_1) - R(t_2)}{(t_2 - t_1) \times R(t_1)} \]

\[ \lambda = \frac{0.85 - 0.2}{60 \times 0.85} = \frac{0.65}{51} = 0.013 \text{ Failures per day} \]
Mean Time Between Failure (MTBF)

- MTBF is useful in safety-critical applications (e.g. avionics, air traffic control, weapons, etc)
- The US government mandates that new air traffic control systems must not be unavailable for more than 30 seconds per year

\[
MTBF = \frac{1}{\lambda}
\]
Consider our previous example where we calculated the failure rate ($\lambda$) of a system to be 0.013. Calculate the MTBF for that system.

$$MTBF = \frac{1}{\lambda}$$

$$= 76.9 \text{ days}$$

This system is expected to fail every 76.9 days.
McCabe’s **Cyclomatic Complexity Metric**

- Complexity is an important attribute to measure
- Measuring Complexity helps us
  - Predict testing effort
  - Predict defects
  - Predict maintenance costs
  - Etc
- Cyclomatic Complexity Metric was designed by McCabe in 1976
- Aimed at indicating a program’s testability and understandability
- It is based on graph theory
- Measures the number of linearly independent paths comprising the program
McCabe’s **Cyclomatic Complexity** Metric

The formula of cyclomatic complexity is:

$$M = V(G) = e - n + 2p$$

where

- $V(G) =$ cyclomatic number of Graph G
- $e =$ number of edges
- $n =$ number of nodes
- $p =$ number of unconnected parts of the graph
Example: Cyclomatic Complexity

Consider the following flowchart...

Calculating cyclomatic complexity

\[ e = 7, \ n=6, \ p=1 \]

\[ M = 7 - 6 + (2 \times 1) = 3 \]
McCabe’s Cyclomatic Complexity

- Note that the number delivered by the cyclomatic complexity is equal to the number of different paths which the program can take.

- Cyclomatic Complexity is additive. i.e. \( M(G_1 \text{ and } G_2) = M(G_1) + M(G_2) \)

- To have good testibility and maintainability, McCabe recommends that no module have a value greater than 10.

- This metric is widely used and accepted in industry.
Halstead’s Software Science

- Halstead (1979) distinguished software science from computer science
- **Premise**: Any programming task consists of selecting and arranging a finite number of program “tokens”
- Tokens are basic syntactic units distinguishable by a compiler
- Computer Program: A collection of tokens that can be classified as either operators or operands
Halstead’s Software Science (2/3)

- Halstead (1979) distinguished software science from computer science
- Primitives:
  - \( n_1 = \) # of distinct operators appearing in a program
  - \( n_2 = \) # of distinct operands appearing in a program
  - \( N_1 = \) total # of operator occurrences
  - \( N_2 = \) total # of operand occurrences
- Based on these primitive measures, Halstead defined a series of equations
<table>
<thead>
<tr>
<th>Measure</th>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocabulary ((n))</td>
<td>(n = n_1 + n_2)</td>
<td></td>
</tr>
<tr>
<td>Length ((N))</td>
<td>(N = N_1 + N_2)</td>
<td></td>
</tr>
<tr>
<td>Volume ((V))</td>
<td>(V = N \log_2(n))</td>
<td>#bits required to represent a program</td>
</tr>
<tr>
<td>Level ((L))</td>
<td>(L = V^* / V)</td>
<td>Measure of abstraction and therefore complexity</td>
</tr>
<tr>
<td>Difficulty ((D))</td>
<td>(D = N/N^*)</td>
<td></td>
</tr>
<tr>
<td>Effort ((E))</td>
<td>(E = V/L)</td>
<td></td>
</tr>
<tr>
<td>Faults ((B))</td>
<td>(B = V/S^*)</td>
<td></td>
</tr>
</tbody>
</table>

Where:
- \(V^* = 2 + n_2 \times \log_2(2 + n_2)\)
- \(M^* = \text{average number of decisions between errors (3000 according to Halstead)}\)
Other useful product metrics

- Cost per function point
- Defects generated per function point
- Percentage of fixes which in turn have defects
Process Metrics
Why measure the process?

- The process creates the product
- If we can improve the process, we indirectly improve the product
- Through measurement, we can understand, control and improve the process
- This will lead to us engineering quality into the process rather than simply taking product quality measurements when the product is done
- We will look briefly at a number of process metrics
Defect Density During Machine Testing

- Defect rate during formal testing is usually positively correlated with the defect rate experienced in the field.
- Higher defect rates found during testing is an indicator that higher defect rates will be experienced in the field.
- **Exception:** In the case of exceptional testing effort or more effective testing methods being employed.
- It is useful to monitor defect density metrics of subsequent releases of the same product.
- In order to appraise product quality, consider the following scenarios.
**Scenario 1:** Defect rate during testing is the same or lower than previous release.

**Reasoning:** Does the testing for the current release deteriorate?

- **No:** Quality Prospect is positive
- **Yes:** You need to perform more testing
Defect Density During Machine Testing

**Scenario 2:** Defect rate is substantially higher than that of the previous release

**Reasoning:** Did we plan for and actually improve testing effectiveness?

- Yes
  - Quality Prospect is positive
- No
  - Quality prospect negative. Perform more testing.
Overall defect density during testing is a summary indicator.

However, the pattern of defect arrivals gives more information.

Even with the same overall defect rate during test, arrival patterns can be different.
Two Different Arrival Patterns

<table>
<thead>
<tr>
<th>WEEK</th>
<th>Defect Arrival Rate</th>
<th>Cumulative Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Defect Arrival Rate Graph](image1)

![Cumulative Rate Graph](image2)

![Defect Arrival Rate Graph](image3)

![Cumulative Rate Graph](image4)
Interpretting Defect Arrival Patterns

- Always look for defect arrivals stabilising at a very low level.
- If they do not stabilise at a low rate, risking the product will be very risky.
- Also keep track of defect backlog over time. It is useless detecting defects if they are not fixed and the system re-tested.
Phase-Based Defect Removal Pattern

- An extension of the defect density metric
- Tracks defects at all phases of the lifecycle
- The earlier defects are found, the cheaper they are to fix
- This metric helps you monitor when your defects are being found
Phase-Based Defect Removal Pattern Example

**Project A**
Most defects found before testing
Ideal situation

**Project B**
Most defects found during testing
More expensive to fix
Should be corrected
Other useful process metrics

- Fix response time
  - Average time to fix a defect
- Percent delinquent fixes
  - Fixes which exceed the recommended fix time according to their severity level
- Fix quality
  - Percentage of fixes which turn out to be defective
People Metrics
Why measure people?

People metrics are of interest to management for:

- Financial purposes (e.g. Putting Joe on project A will cost me Lm500 per function point)
- Project management purposes (e.g. Michele needs to produce 5 function points per day in order to be on time)
- HR problem identification (e.g. On average, developers produce 5 defects per hour. James produces 10. Why?)
Warning on People Measurement

- People do not like being measured
- In many cases, you will not be able to look at numbers and draw conclusions.
- For example, at face value, Clyde may take a longer time to finish his work when compared to colleagues. However, further inspection might reveal that his code is bug free whilst that of his colleagues needs a lot of reworking.
- Beware when using people metrics. Only use them as indicators for potential problems.
- You should never take disciplinary action against personell based simply on people metrics.
Some people metrics…

For individual developers or teams:

- Cost per Function Point
- Mean Time required to develop a Function Point
- Defects produced per hour
- Defects produced per function point
Object Oriented Design Metrics
Why measure OO Designs?

- OO has become a very popular paradigm
- Measuring the Quality of a design helps us identify problems early on in the life cycle
- A set of OO Design metrics were proposed by Chidamer and Kemerer (MIT) in 1994.

![Cost ($) vs. Phase](chart.png)
Unique OO Characteristics (1/2)

- **Encapsulation**
  - Binding together of a collection of items
    - State information
    - Algorithms
    - Constants
    - Exceptions
    - ...

- **Abstraction and Information Hiding**
  - Suppressing or hiding of details
  - One can use an object’s advertised methods without knowing exactly how it does its work
Unique OO Characteristics (2/2)

- **Inheritance**
  - Objects may acquire characteristics of one or more other objects
  - The way inheritance is used will affect the overall quality of a system

- **Localisation**
  - Placing related items in close physical proximity to each other
  - In the case of OO, we group related items into objects, packages, etc.
Class

- Template from which objects are created
- Class design affects overall:
  - Understandability
  - Maintainability
  - Testability
- Reusability is also affected by class design
  - E.g. Classes with a large number of methods tend to be more application specific and less reusable
Measurable Structures in OO (2/5)

- **Message**
  - A request made by one object to another object
  - Receiving object executes a method
  - It is important to study message flow in an OO system
    - Understandability
    - Maintainability
    - Testability
  - The more complex message flows between objects are, the less understandable a system is
**Coupling**
- A measure of the strength of association established by connections between different entities
- Occurs through:
  - Use of an object’s methods
  - Inheritance


**Cohesion**

- The degree to which methods in a class are related to each other
- Effective OO designs **maximise cohesion** because they promote encapsulation
- A high degree of cohesion indicates:
  - Classes are self contained
  - Fewer messages need to be passed (more efficiency)
Measurable Structures in OO (5/5)

- **Inheritance**
  - A mechanism which allows an object to acquire the characteristics of one or more other objects
  - Inheritance can **reduce complexity** by reducing the number of methods and attributes in child classes
  - Too much inheritance can make the system **difficult to maintain**
Consider the class \( C \) with methods \( m_1, m_2, \ldots m_n \).

Let \( c_1, c_2 \ldots c_n \) be the complexity of these methods.

\[
WMC = \sum_{i=1}^{n} c_i
\]
Weighted Methods Per Class (WMC)

- Refers to the complexity of an object
- The number of methods involved in an object is an indicator of how much time and effort is required to develop
- Complex classes also make their child classes complex
- Objects with large number of methods are likely to be more application-specific and less reusable
- Guidelines: WMC of 20 for a class is good but do not exceed 40.
- Affects:
  - Understandability, Maintainability, Reusability
Weighted Methods Per Class (WMC)
Depth of Inheritance Tree (DIT)

- The Depth of Inheritance of a class is its depth in the inheritance tree.
- If multiple inheritance is involved, the DIT of a class is the maximum distance between the class and the root node.
- The root class has a DIT of 0.
Dept of Inheritance Tree (DIT)
Depth of Inheritance Tree (DIT)

- The deeper a class is in the hierarchy, the greater the number of methods likely to inherit from parent classes – *more complex*
- Deeper trees → More Reuse
- Deeper trees → Greater Design Complexity
- DIT can analyse efficiency, reuse, understandability and testability
Depth of Inheritance Tree (DIT)
Number of Children (NOC)

- A simple metric
- Counts the number of immediate subclasses of a particular class
- It is a measure of how many subclasses are going to inherit attributes and methods of a particular class
Number of Children (NOC)

- Generally it is better to have depth than breadth in the class hierarchy
  - Promotes reuse of methods through inheritance
- Classes higher up in the hierarchy should have more subclasses
- Classes lower down should have less
- The NOC metric gives an indication of the potential influence of a class on the overall design
- Attributes: Efficiency, Reusability, Testability
Number of Children (NOC)
Another simple metric

CBO for a particular class is a count of how many non-inheritance related couples it maintains with other classes
Coupling Between Objects (CBO)

- Excessive coupling outside inheritance hierarchy:
  - Detrimental to modular design
  - Prevents reuse
- The more independent an object is, the easier it is to reuse
- Coupling is **not** transitive
- The more coupling there is in a design, the more sensitive your system will be to changes
- More coupling \(\rightarrow\) More Testing
- **Rule of thumb**: Low coupling but high cohesion
Coupling Between Objects (CBO)
Response for a Class (RFC)

\[ \text{RFC} = |RS| \]

where RS is the *response set* of a class

\[ RS = \{M_i\} \cup \{R_i\} \]

- \( M_i \) = All the methods in a class
- \( R_i \) = All methods called by that class
Response for a Class (RFC)

- If a large number of methods can be invoked in response to a message, testing and debugging becomes more complicated.

- More methods invoked → More Complex Object

- **Attributes**: understandability, maintainability, testability
Response for a Class (RFC)
Lack of Cohesion in Methods (LCOM)

- Consider a class $C_1$ with methods $M_1$, $M_2$, … $M_n$
- Let $\{I_i\}$ be the set of instance variables used by methods $M_i$
- There are $n$ such sets: $\{I_1\}$, $\{I_2\}$, … $\{I_n\}$

$\text{LCOM} = \text{The number of disjoint sets formed by the intersection of the } n \text{ sets}$
Lack of Cohesion in Methods (LCOM)

- Cohesiveness of methods within a class is desirable
  - Promotes Encapsulation
- Lack of cohesion implies that a class should be split into 2 or more classes
- This metric helps identify flaws in a design
- Low Cohesion $\Rightarrow$ Higher Complexity
Lack of Cohesion in Methods (LCOM)
Conclusions

- We have examined a set of metrics which allow you to analyse the quality of OO Designs

- Thresholds:
  - We can provide guidelines
  - However, each project may have different needs

- When possible, try to plot metric results on graphs.
  - Easier to interpret
Conclusions

- Many other metrics exist and measure
  - Different quality attributes
  - Different types of systems
  - Different process attributes
  - Different people attributes

- Beyond the scope of this short course
Conclusions

As a result of this course, we hope this that you now:

- Appreciate the uses of measurement in general and the need to apply it to software
- Have a good idea of what steps may be involved in setting up a measurement programme
- Know a few metrics which you can use in the industry
- Understand OO metrics in some detail and are able to interpret them in practice
Conclusions

- Watch out for a metrics assignment covering 30% of your marks