# Software Measurement

Mark Micallef
mmica01@um.edu.mt

# Brief Course Overview

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

# Introduction to Measurement Theory

# What is measurement?

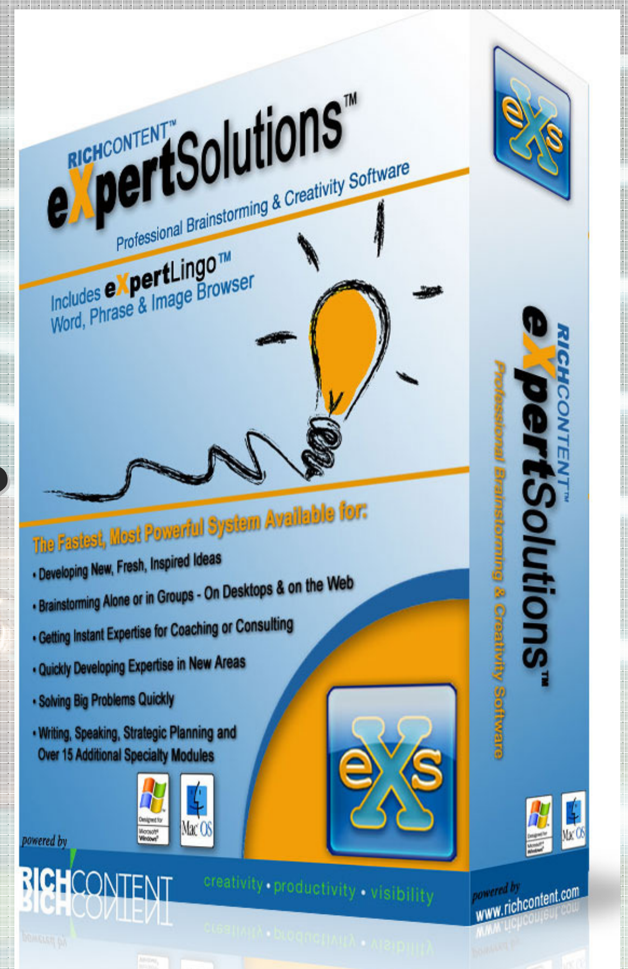
Measurement is the process by which numbers or symbols are assigned to attributes of entities in the world *according to clearly defined rules*.

# 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

Theory

↓

| | | |
|---|---|---|
| Abstract World | 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.<br><br>Proof or refute the hypothesis. |

# Example: Proving a theory

**Abstract World**

**Theory** — Adding more developers to a late project will only make it later.

↓

**Proposition** —
- 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.

---

**Empirical World**

↓

**Hypothesis** —
- 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)

↓

**Data Analysis** — 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.
- ***Hyptohesis*** – 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*** – Characterisics / 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 (1/2)

**Example:** ***A religion nominal scale***

| | |
|---|---|
| Joe | Michelle |
| Rachel | Christine |
| Michael | James |
| Clyde | Wendy |

**Catholic**

**Muslim**

**Other**

**Jewish**

# The Nominal Scale (2/2)

- The most simple measurment 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 (1/2)

**Example:** ***A <u>degree-classification</u> ordinal scale***

| | |
|---|---|
| Joe | Michelle |
| Rachel | Christine |
| Michael | James |
| Clyde | Wendy |

**1st Class**

**2nd Class**

**3rd Class**

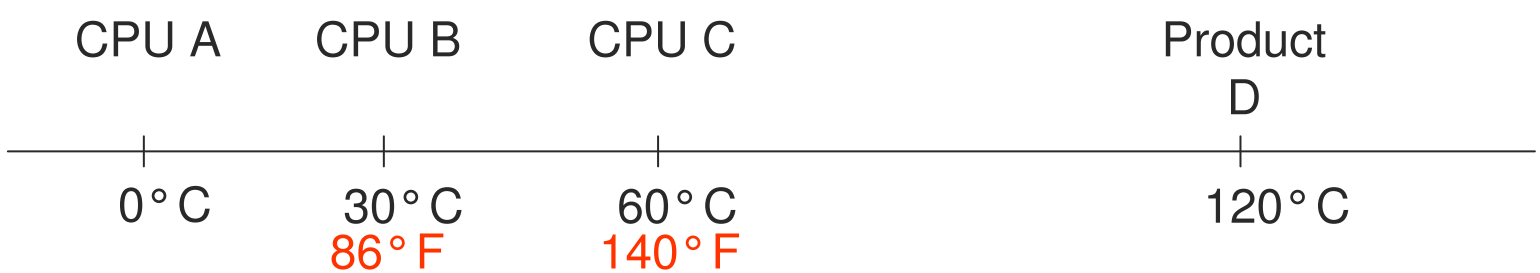
**Failed**

# The Ordinal Scale (2/2)

- Elements classified into categories
- Categories are ranked
- Categories are transitive A > B & B > C ➔ 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



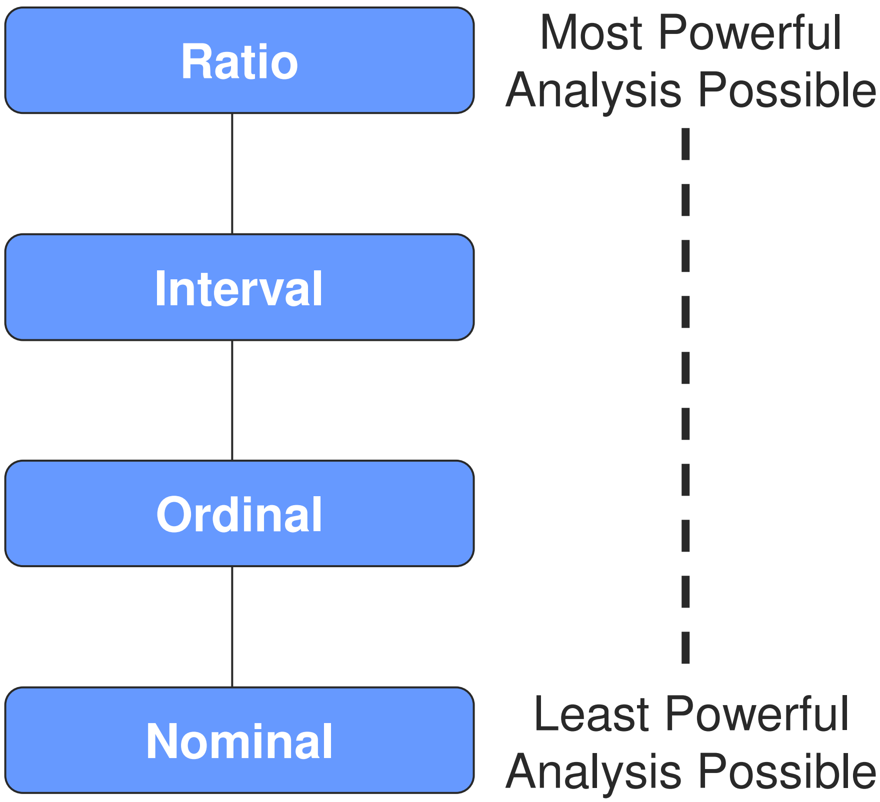

**Temperature of Different CPUs**

# 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

# Measurement Scales Hierarchy

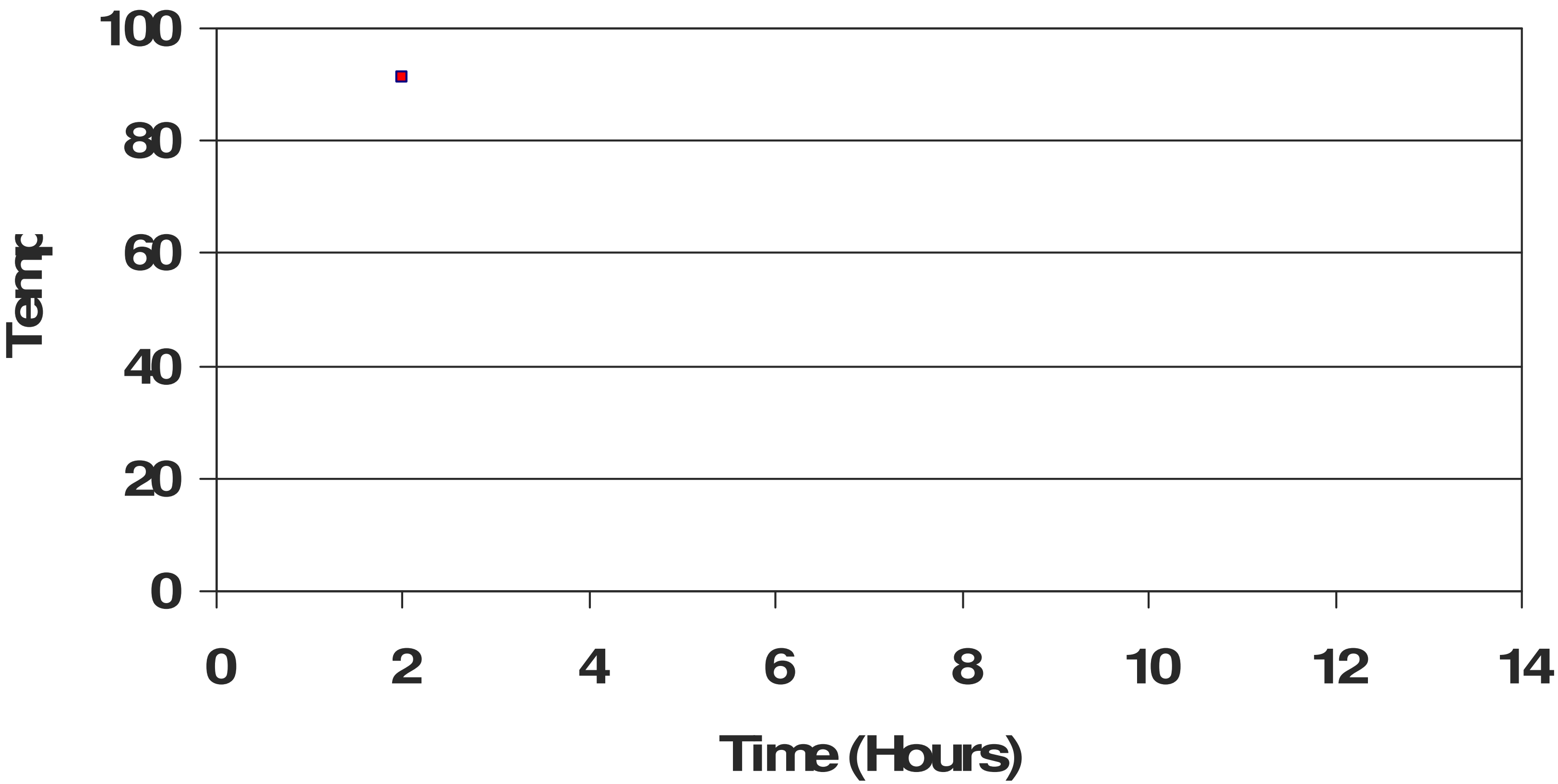
- Scales are hierarchical
- Each higher-level scale possesses all the properties of the lower ones
- A higher-level of measurement can be reduced to a lower one but not vice-versa



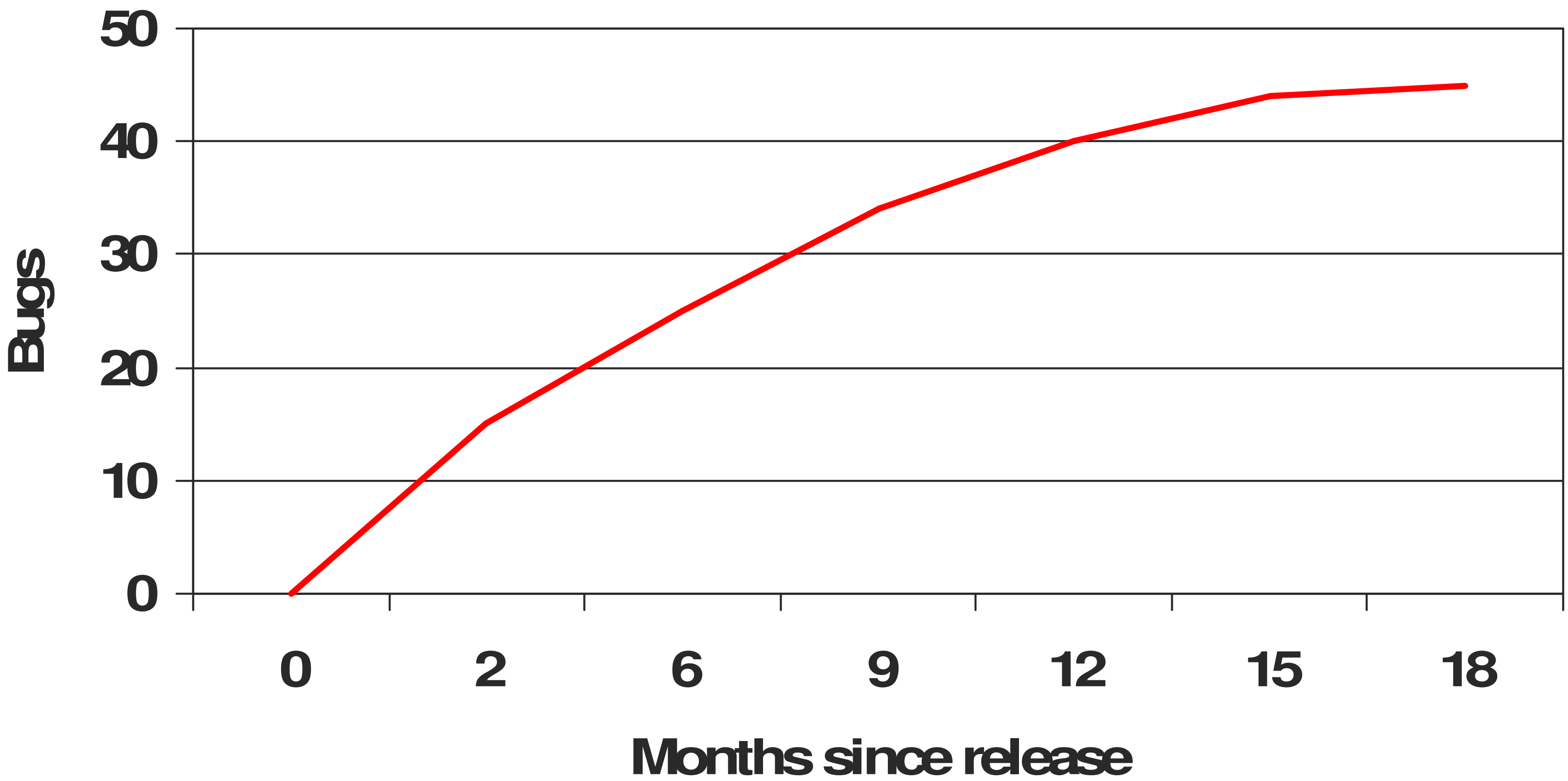
# Measures, Metrics and Indicators

- ***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 particalar 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

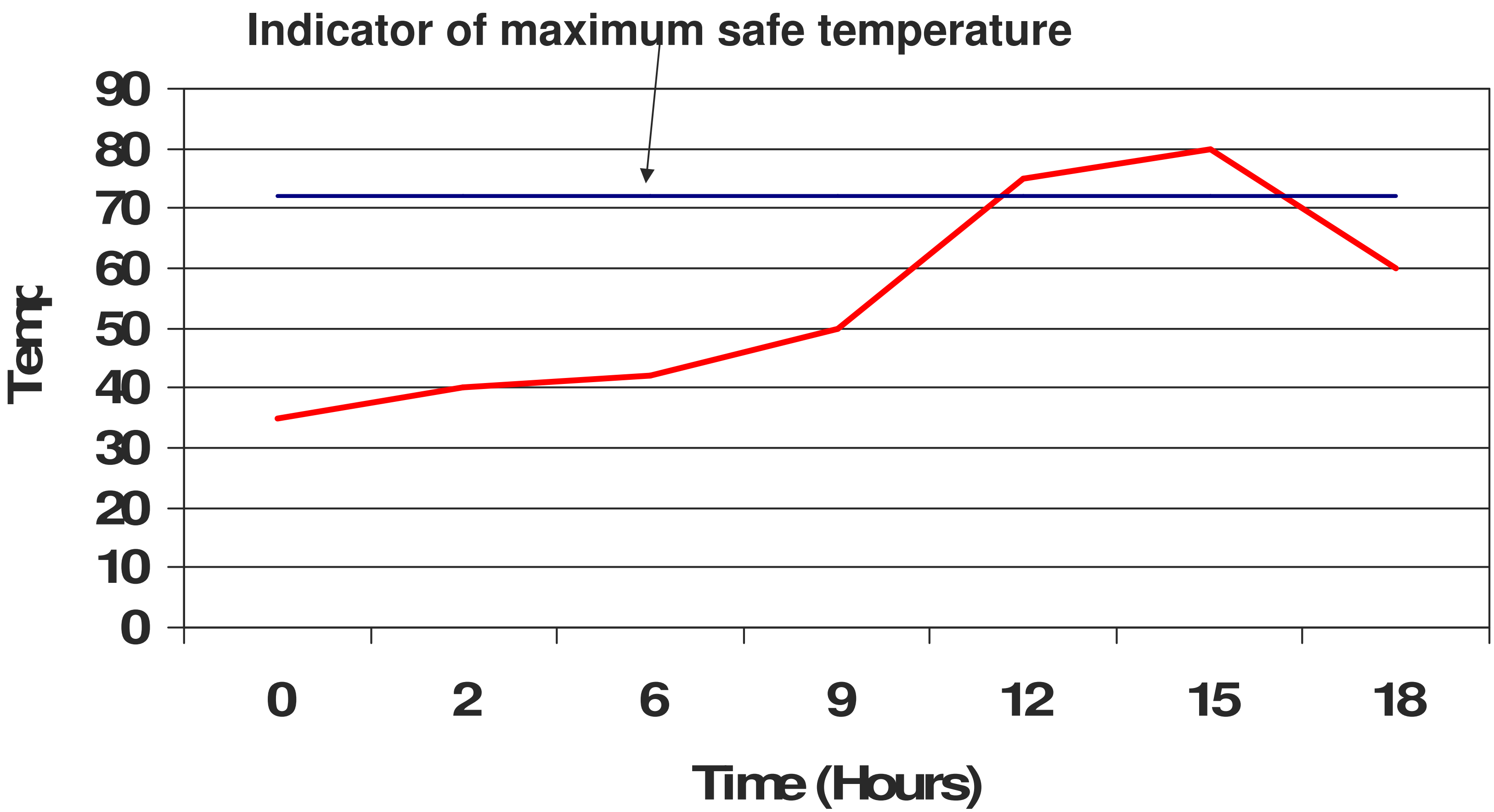
# Example of a Measure



# Example of a Metric



# Example of a Indicator



# 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. $\frac{\text{Number of satisfied customers}}{\text{Total number of customers}}$

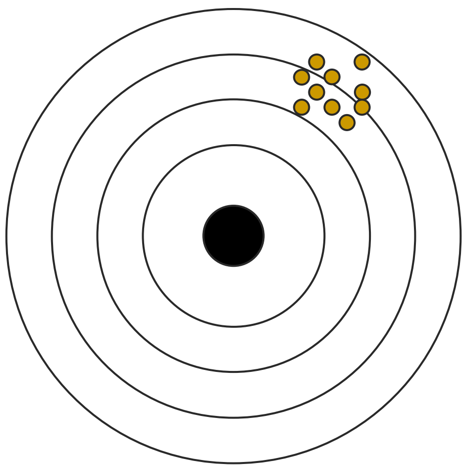
# Some basic measures (2/2)

- 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

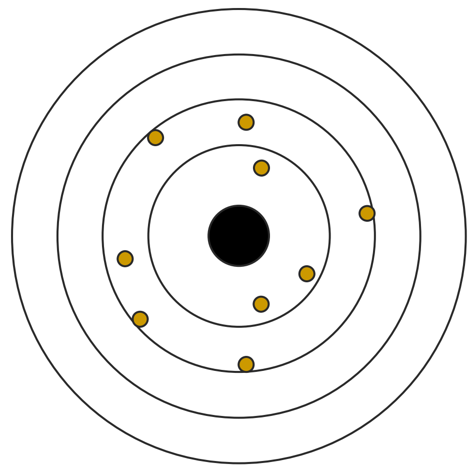
# Reliability and Validity of Measurements

- 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.

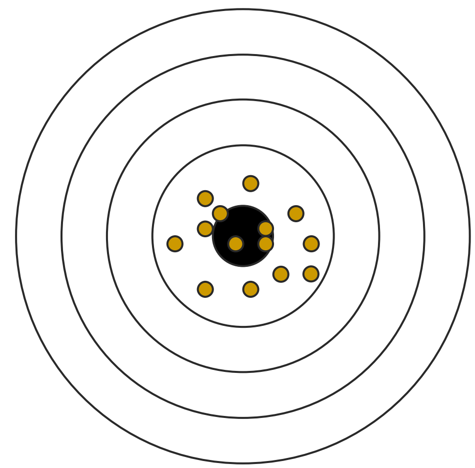
# Reliability and Validity of Measurements



**Reliable but not valid**



**Valid but not reliable**



**Reliable and Valid**

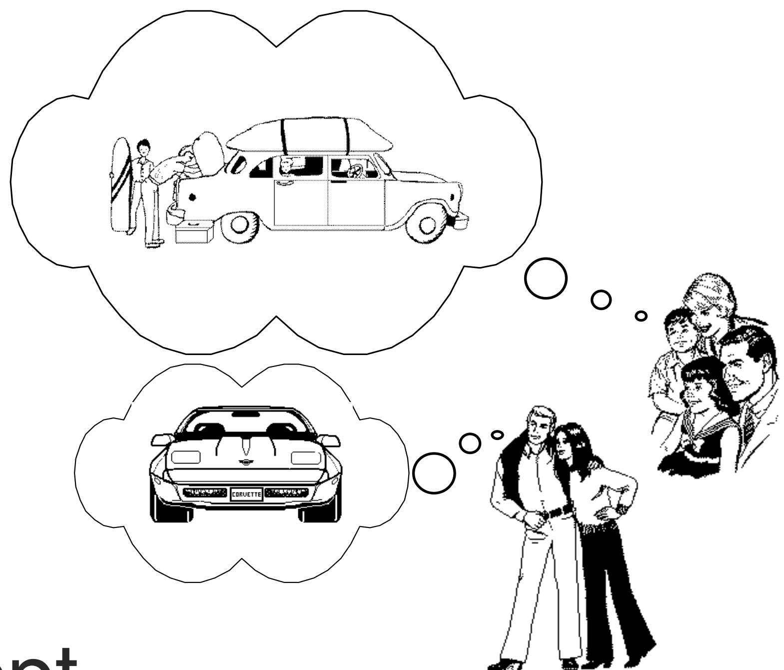
# Measuring Software

# What makes quality software?

- Cheap?
- Reliable?
- Testable?
- Secure?
- Maintainable?
- …

# What makes quality software?

- There is not 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 Measurment

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

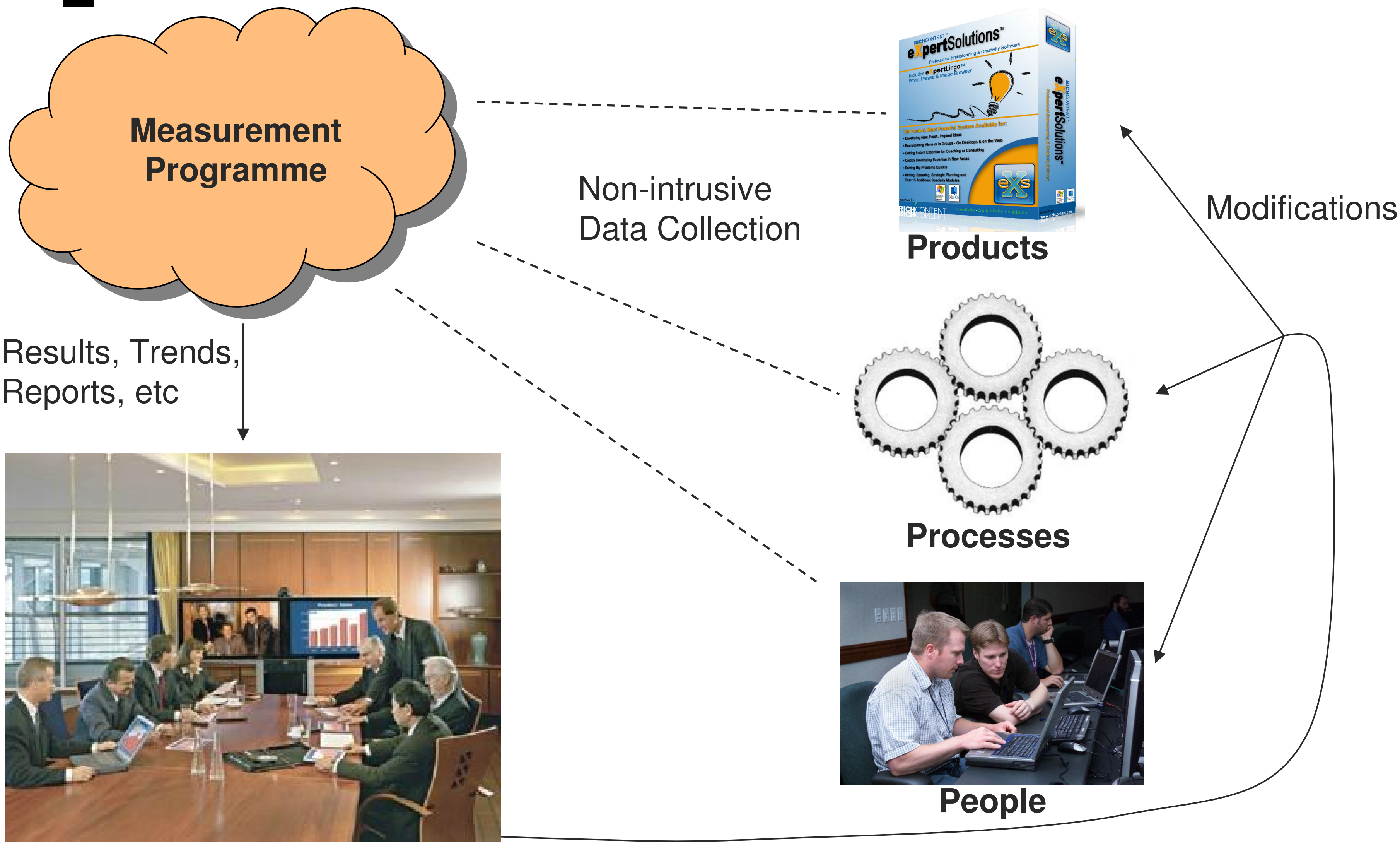
# Measuring the Process

- 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



# 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

# Collecting Software Engineering Data

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 (1/3)

## 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, adherance to schedule, time that problems remain open, and more…

# Examples of Metrics Programmes (2/3)

## 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 (3/3)

## 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

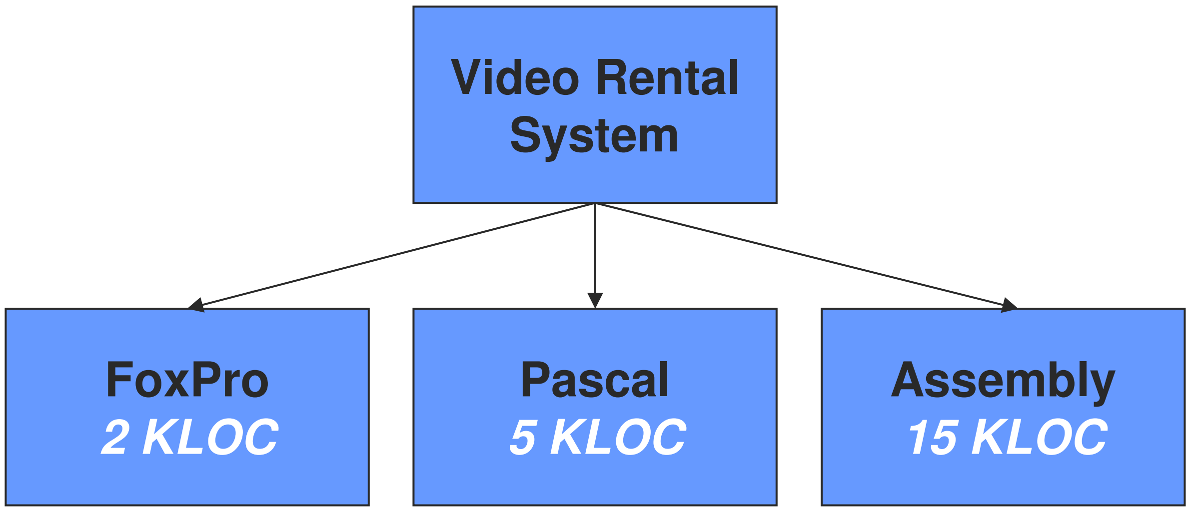
# What can we measure about a product?

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

# Size Metrics

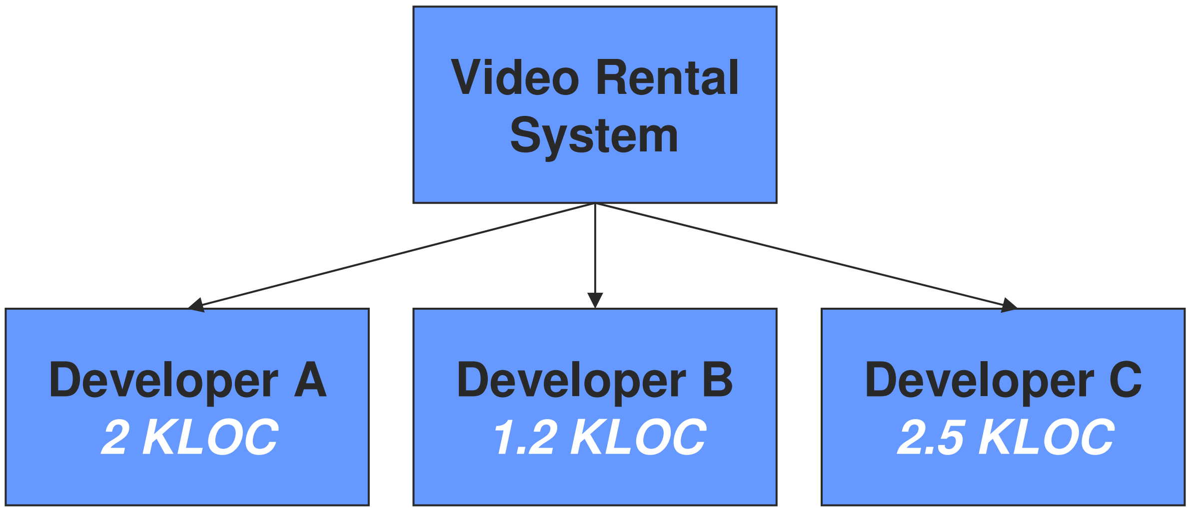
- Knowing the size of a system was important for comparing different systems together
- Software measured in lines of code (LOC)
- As systems grew larger KLOC (thousands of lines of code) was also used

# The problems with LOC (1/3)




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

# The problems with LOC (2/3)




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

# The problems with LOC (3/3)

- To calculate LOC you have to wait until the system is implementet
- 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

# Overview of Function Points (1/3)

- 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 independant

# Overview of Function Points (2/3)

- A function point count is calculated as a wieghted 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*

# Overview of Function Points (3/3)

- The *simplest* way to calculate a function point count is calculated as follows:

  (No. of external inputs x 4) +

  (No. of external outputs x 5) +

  (No. of logical internal files x 10) +

  (No. of external interface files x 7) +

  (No. of external enquiries x 4)

# Function Points Example (1/2)

**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 (2/2)

External Inputs: 1

External Outputs: 2

Logical Internal Files: 1

External Interface Files: 0

External Enquiries: 1

Total Functionality is (1x4) + (2x5) + (1x10) + (0x7) +(1x4) = 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 (1/3)

- **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

# The GSC Function Points Extension (2/3)

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 GSC Function Points Extension (3/3)

- The analyst/software engineer assigns a value between 0 and 5 to each question
- 0 = *not applicable and* 5 = *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:

$$FP = FC \times VAF$$

# GSC Example (1/2)

Consider the bug-reporting system for which we already looked at and suppose the analyst involved answers the GSC questions as follows…

1. Data communications **5**
2. Distributed Functions **0**
3. Performance **1**
4. Heavily used configuration **0**
5. Transaction rate **1**
6. Online Data Entry **5**
7. End-user Efficiency **0**
8. On-line update **3**
9. Complex Processing **1**
10. Reusability **0**
11. Installation ease **2**
12. Operational Ease **3**
13. Multiple sites **4**
14. Facilitation of Change **0**

**Total GSC Score** = **25**

# GSC Example (2/2)

- 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.

$$VAF = 0.65 + (0.01 \times 25) = 0.9$$
$$FC = 28 \times 0.9 = \underline{\mathbf{25.2}}$$

- 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 λ (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

# Example of Failure Rate (1/2)

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 (2/2)

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

$$\lambda = \frac{0.85 - 0.2}{60 \times 0.85}$$

$$= \frac{0.65}{51}$$

$$= 0.013$$ **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

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

# MTBF Example

Consider our previous example where we calculated the failure rate (λ) 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

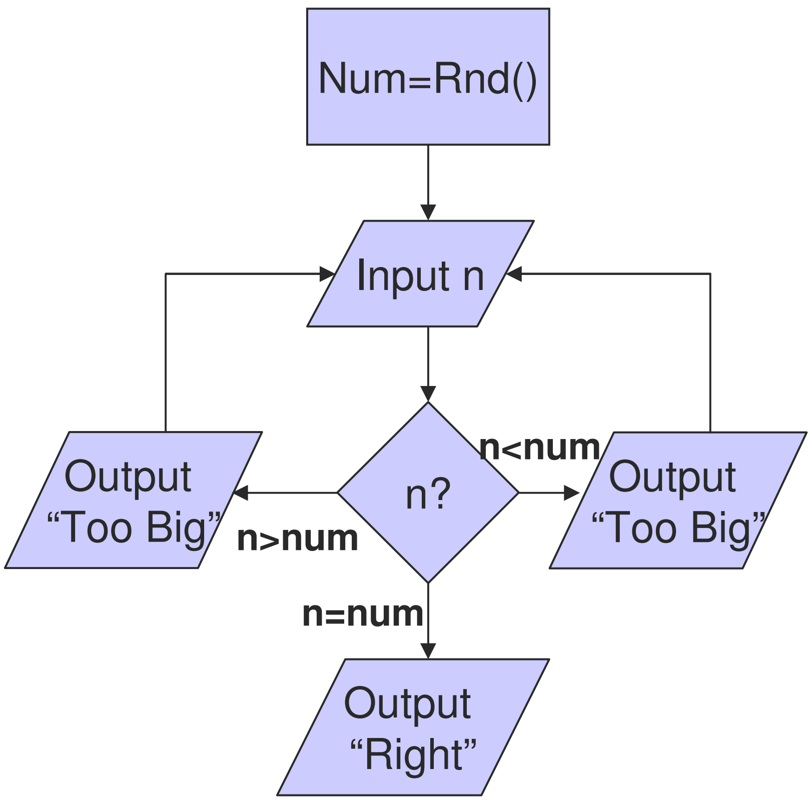
**e** = number of edges

**n** = number of nodes

**p** = number of unconnected parts of the graph

# Example: Cyclomatic Complexity

**Consider the following flowchart…**




**<u>Calculating cyclomatic complexity</u>**

$e = 7, n=6, p=1$

$M = 7 - 6 + (2x1) =$ **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. $M(G_1$ 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 (1/3)

- Halstead (1979) distinguished software science from computer science
- **Premise:** Any programming task consists of selecting and arranging a finite number of progam "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 occurences

  $N_2$ = total # of operand occurences
- Based on these primitive measures, Halstead defined a series of equations

# Halstead's Software Science (3/3)

| | |
|---|---|
| Vocabulary (n) | $n = n_1 + n_2$ |
| Length (N) | $N = N_1 + N_2$ |
| Volume (V) | $V = N \log_2(n)$ ← #bits required to represent a program |
| Level (L) | $L = V^* / V$ ← Measure of abstraction and therefore complexity |
| Difficulty (D) | $D = N/N^*$ |
| Effort (E) | $E = V/L$ |
| Faults (B) | $B = V/S^*$ |

Where:

$V^* = 2 + n_2 \times \log_2(2 + n_2)$

M* = 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

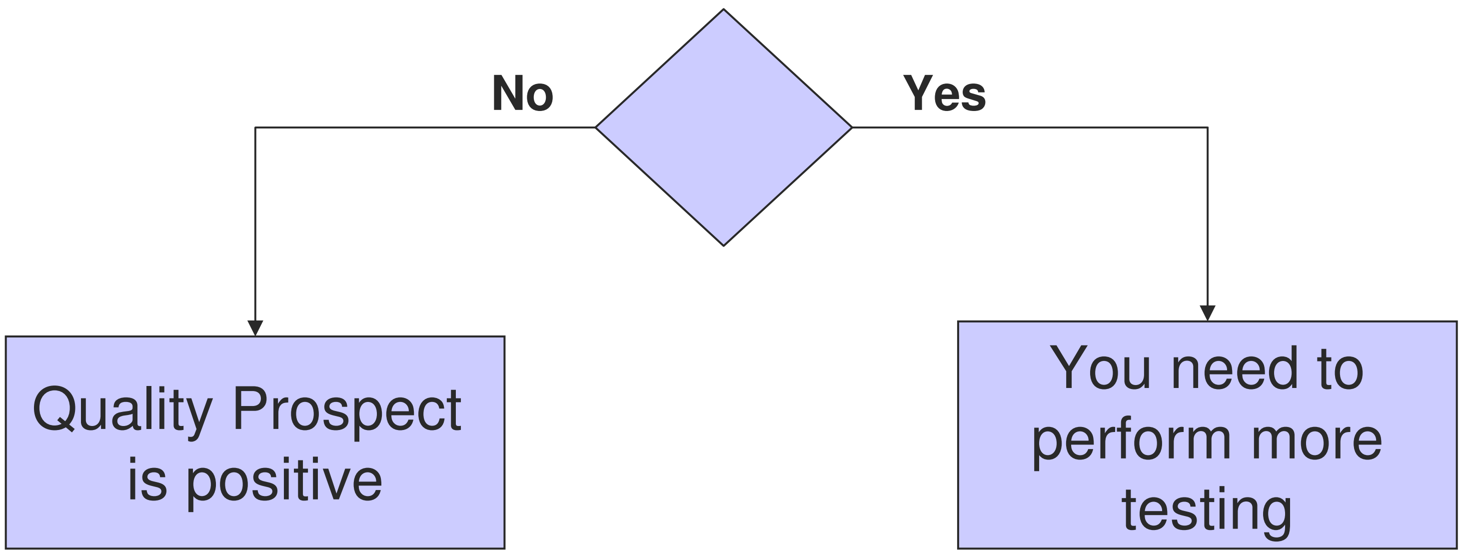
# Defect Density During Machine Testing

- Defect rate during formal testingis 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

# Defect Density During Machine Testing

**Scenario 1:** Defect rate during testing is the same or lower than previous release.

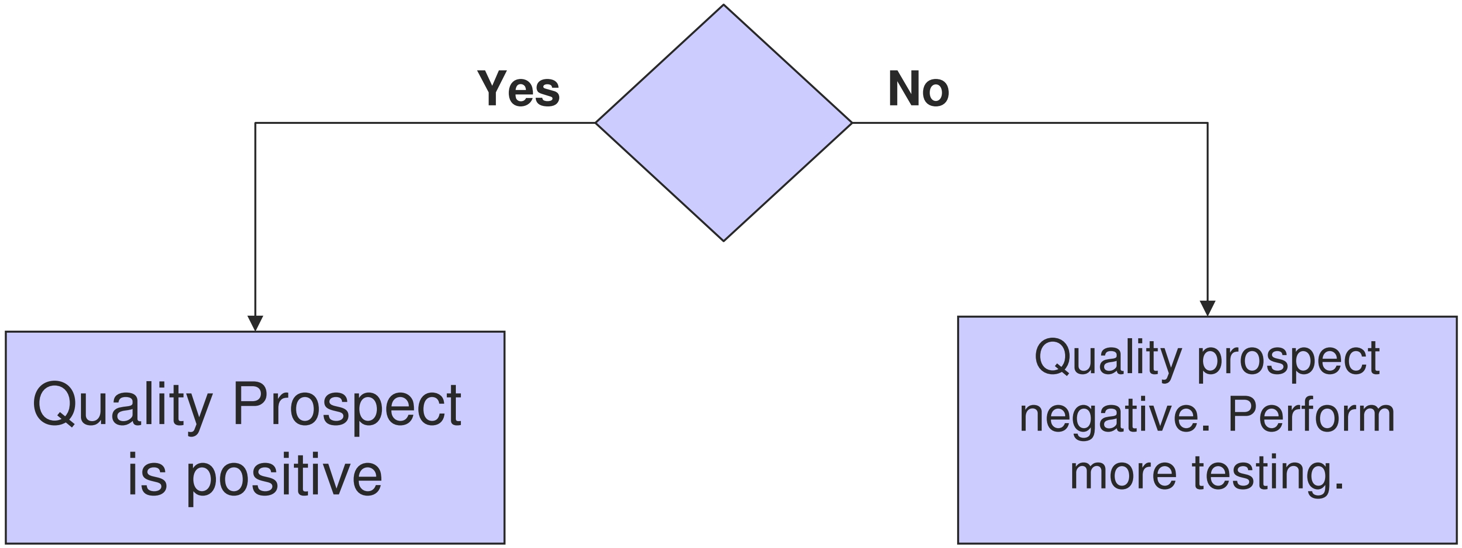
Reasoning: Does the testing for the current release deteriorate?



# Defect Density During Machine Testing

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

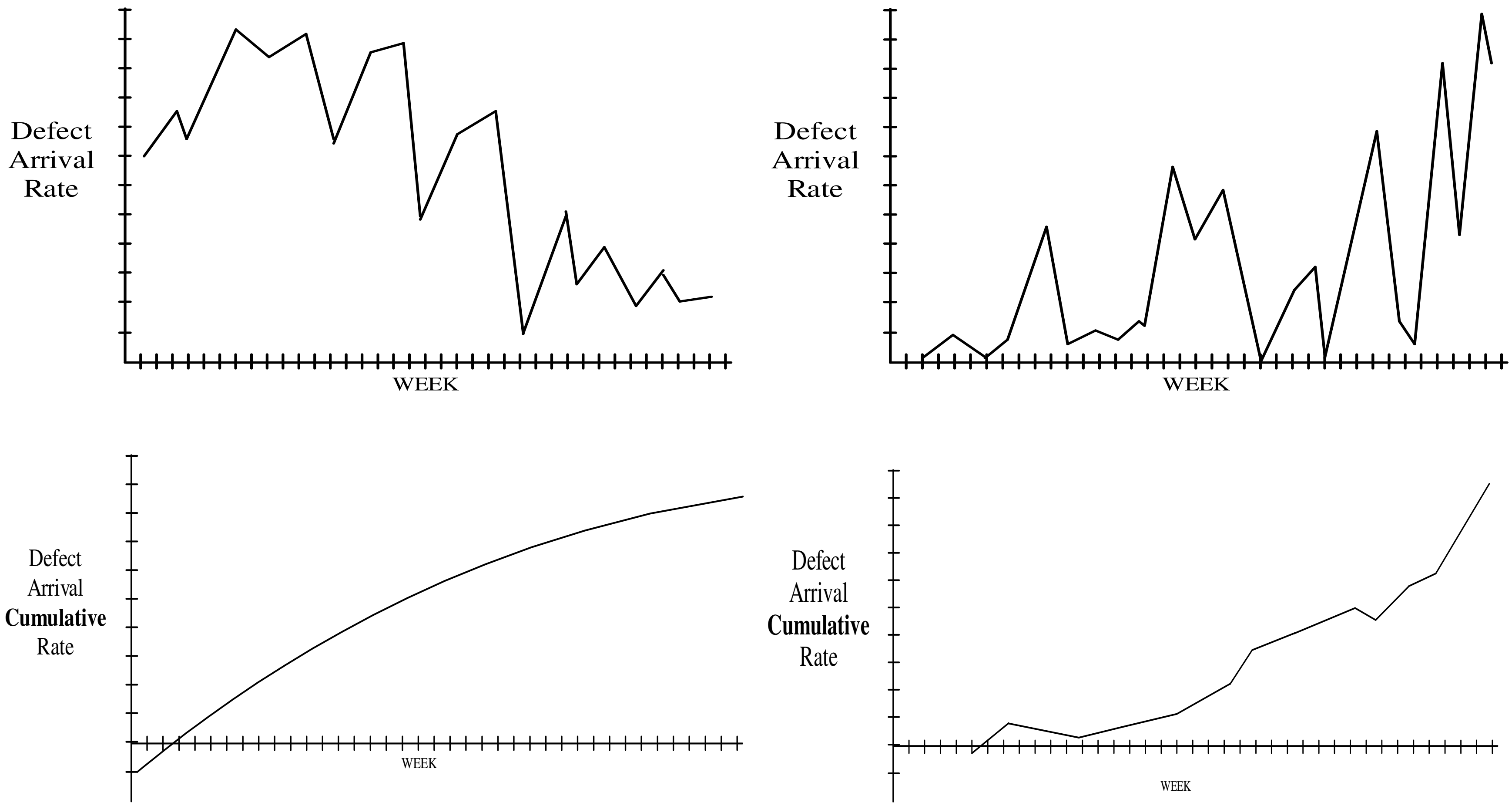
Reasoning: Did we plan for and actually improve testing effectiveness?



# Defect Arrival Pattern During Testing

- Overall defect density during testing is a summary indicator
- However, the patter of defect arrivals gives more information
- Even with the same overall defect rate during test, arrival patterns can be different

# Two Different Arrival Patterns



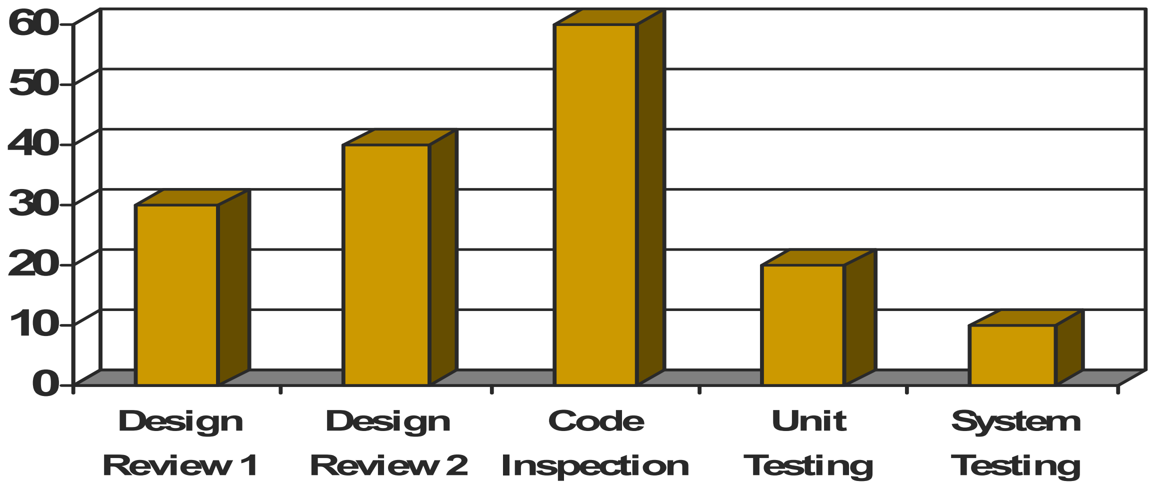
# Interpretting Defect Arrival Patterns

- 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

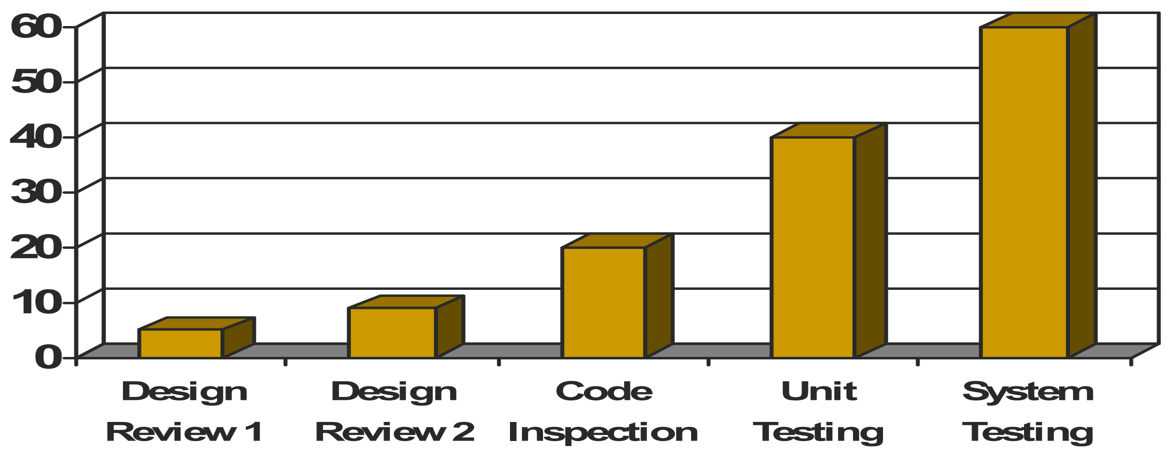
# Phase-Based Defect Removal Pattern Example




**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 a 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…

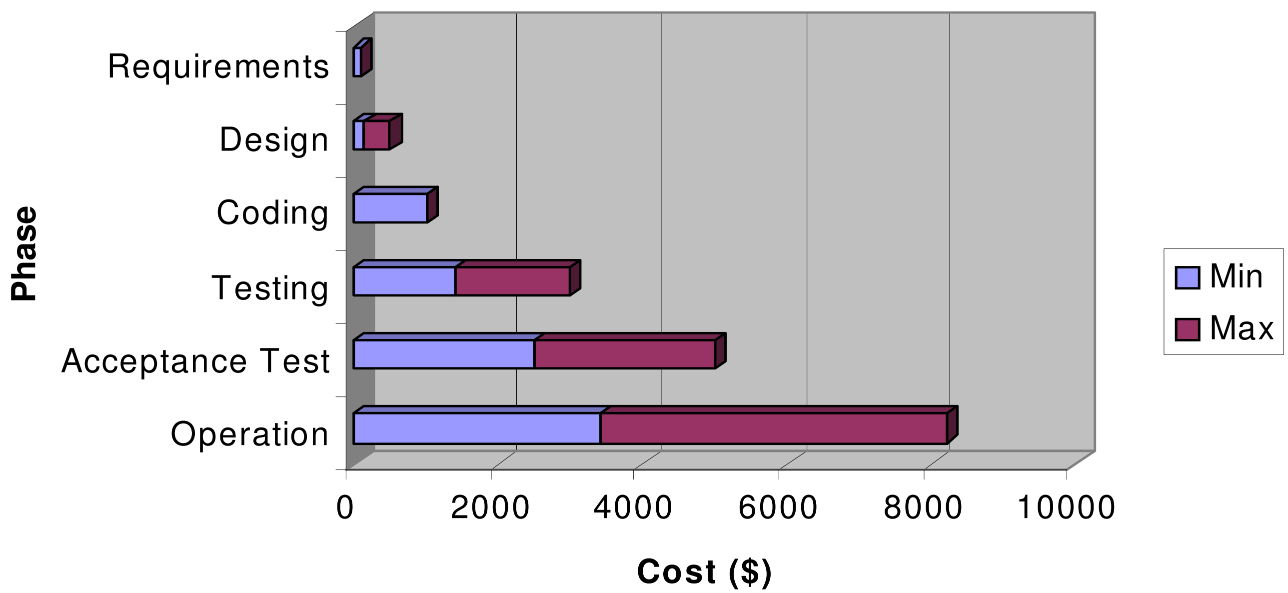
For individual developers or teams:

- 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.



# 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, ets

# Measurable Structures in OO (1/5)

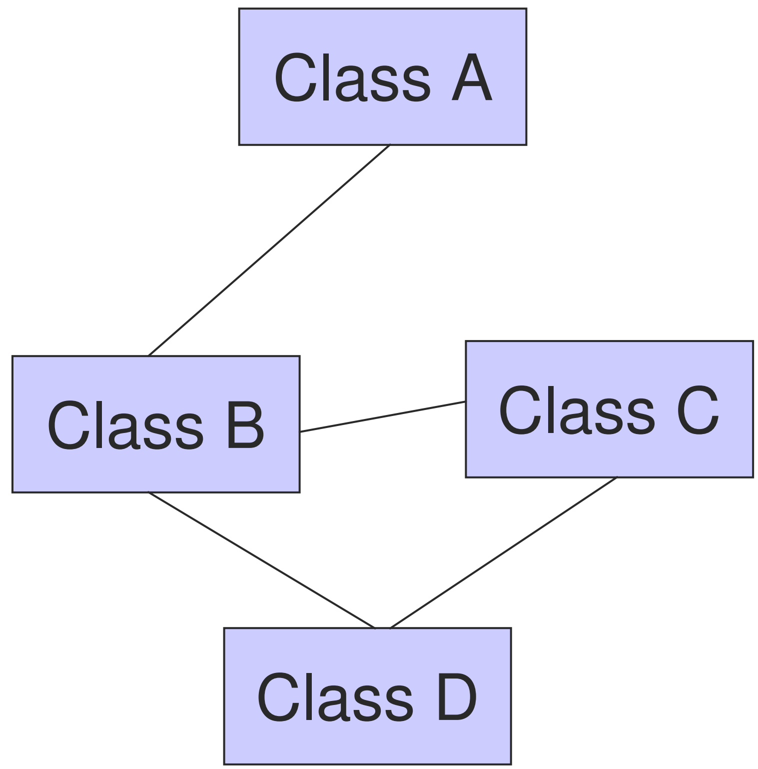
- **Class**
  - Template from which objects are created
  - Class design affects overall:
    - Understandability
    - Maintainability
    - Testability
  - Reusability is also affected by class design
    - E.g. Classes with a large number of methods tend to be more application specific and less reusable

# 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

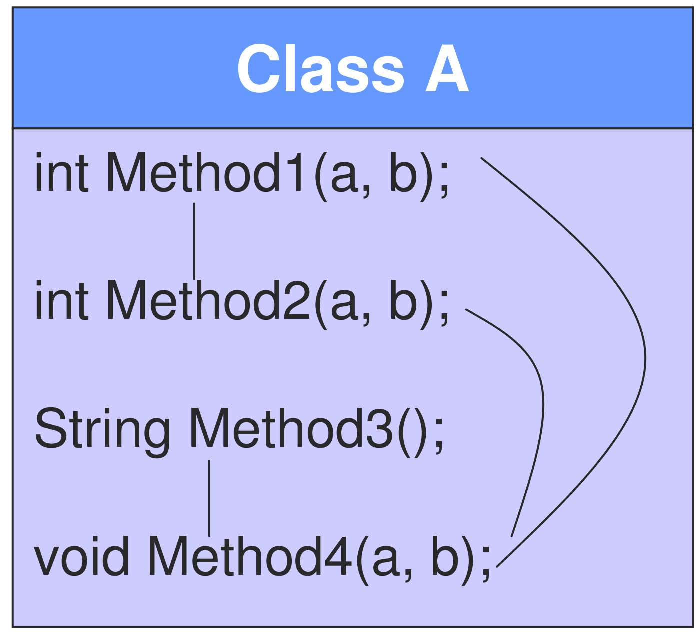
# Measurable Structures in OO (3/5)

- **Coupling**
  - A measure of the strength of association established by connections between different entities
  - Occurs through:
    - Use of an object's methods
    - Inheritance



# Measurable Structures in OO (4/5)

- **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

# Weighted Methods Per Class (WMC)

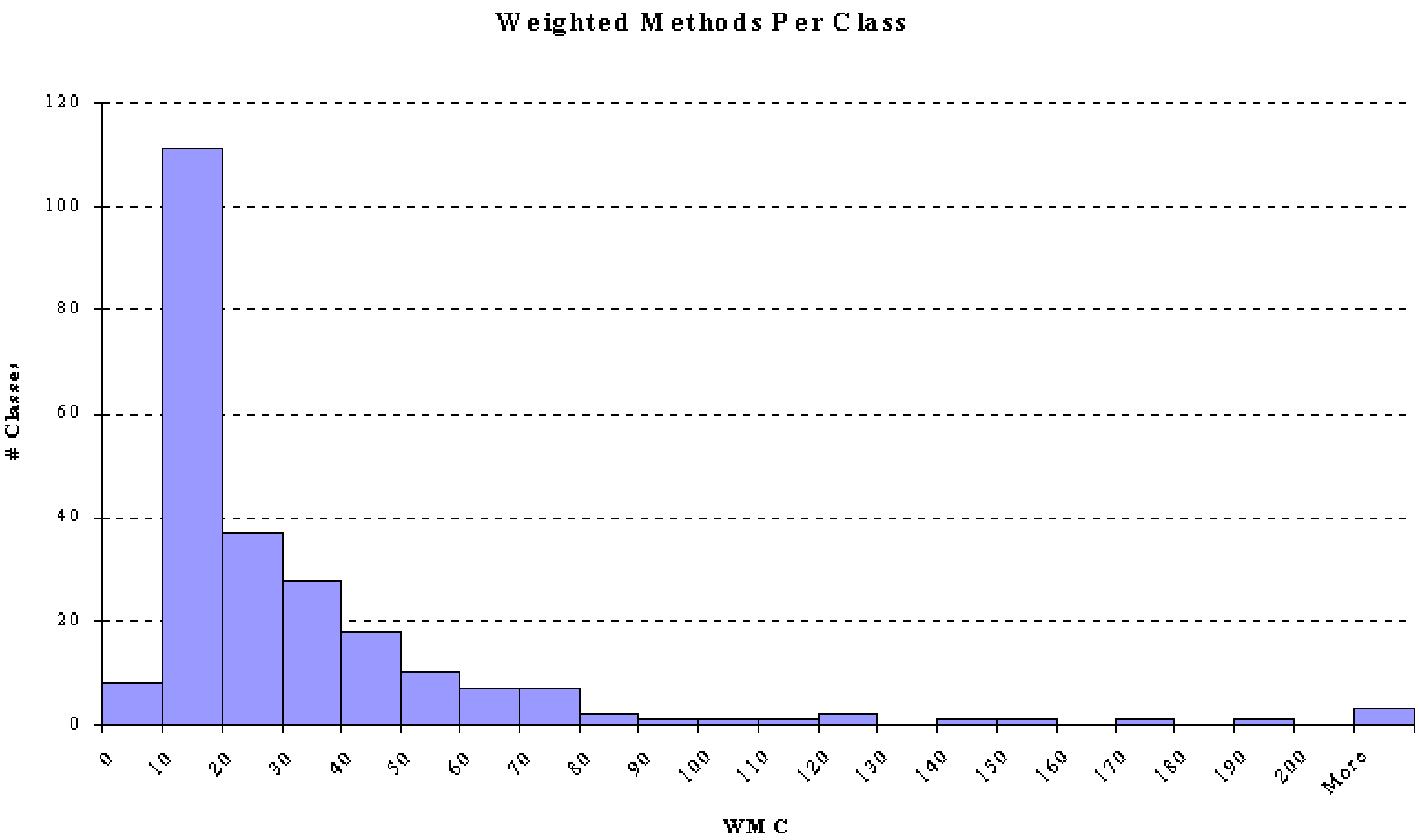
- Consider the class C with methods $m_1$, $m_2$, … $m_{n.}$
- Let $c_1$, $c_2$ … $c_n$ be the complexity of these methods.

$$WMC = \sum_{i=1}^{n} c_i$$

# 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

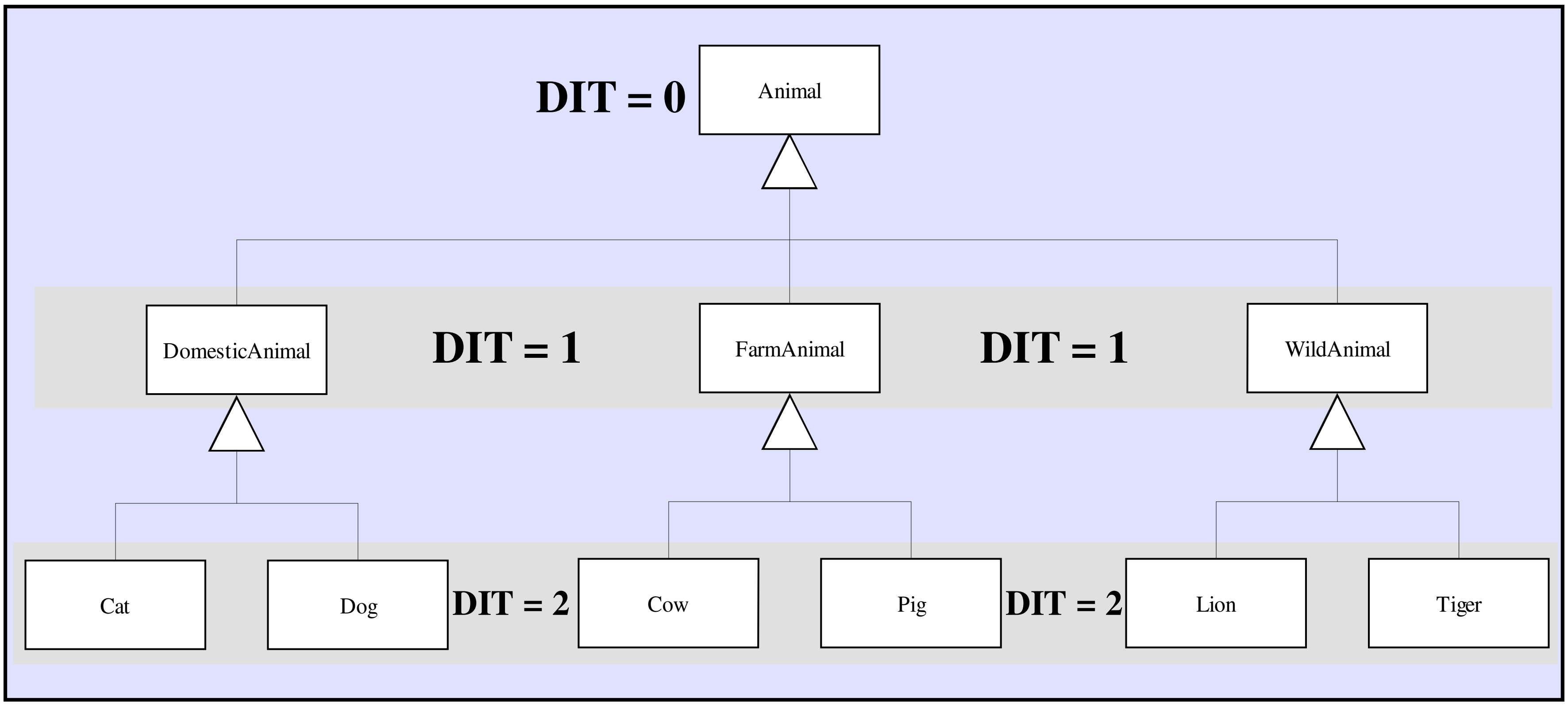
# Weighted Methods Per Class (WMC)



# 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

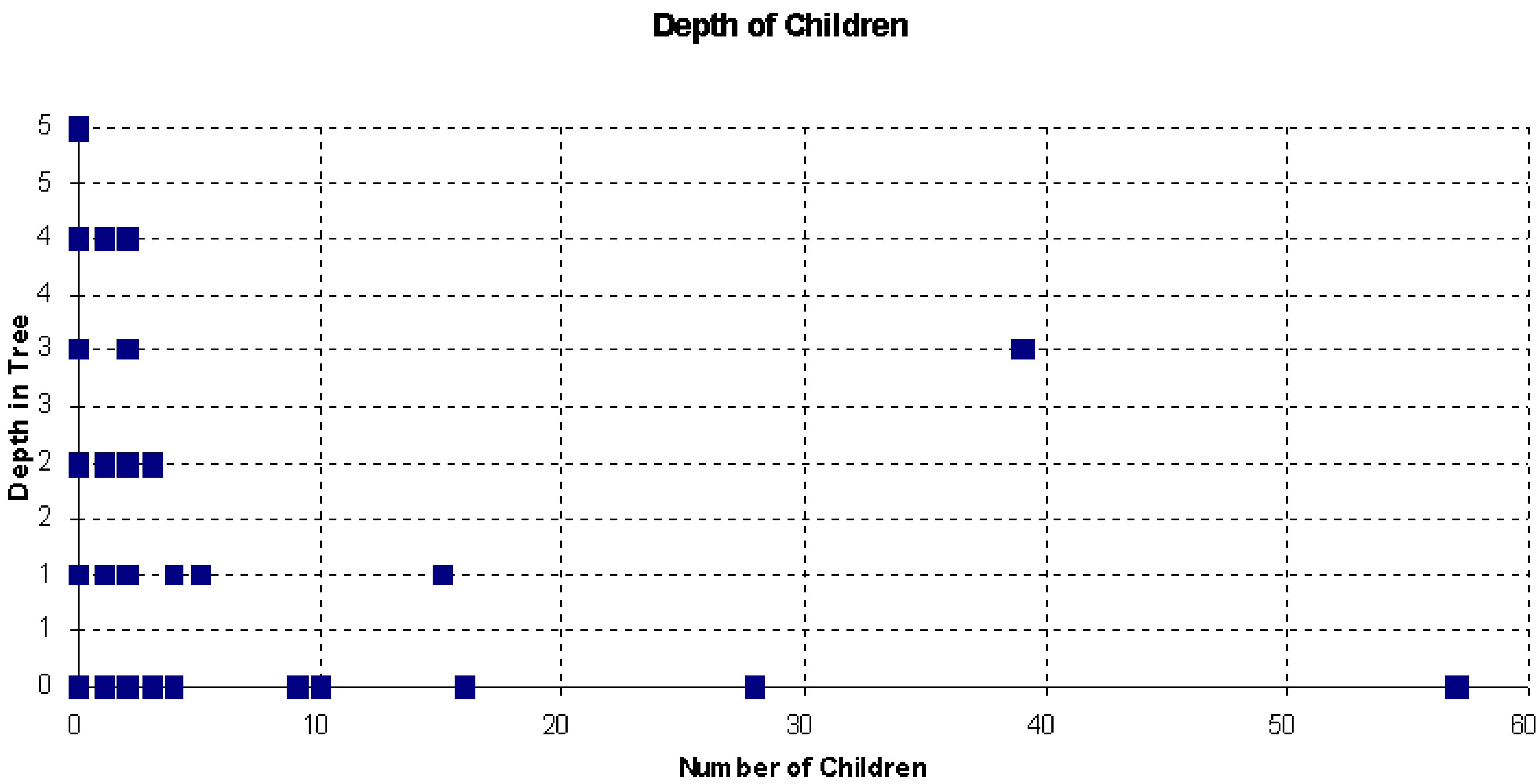
# Dept of Inheritance Tree (DIT)



# 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)



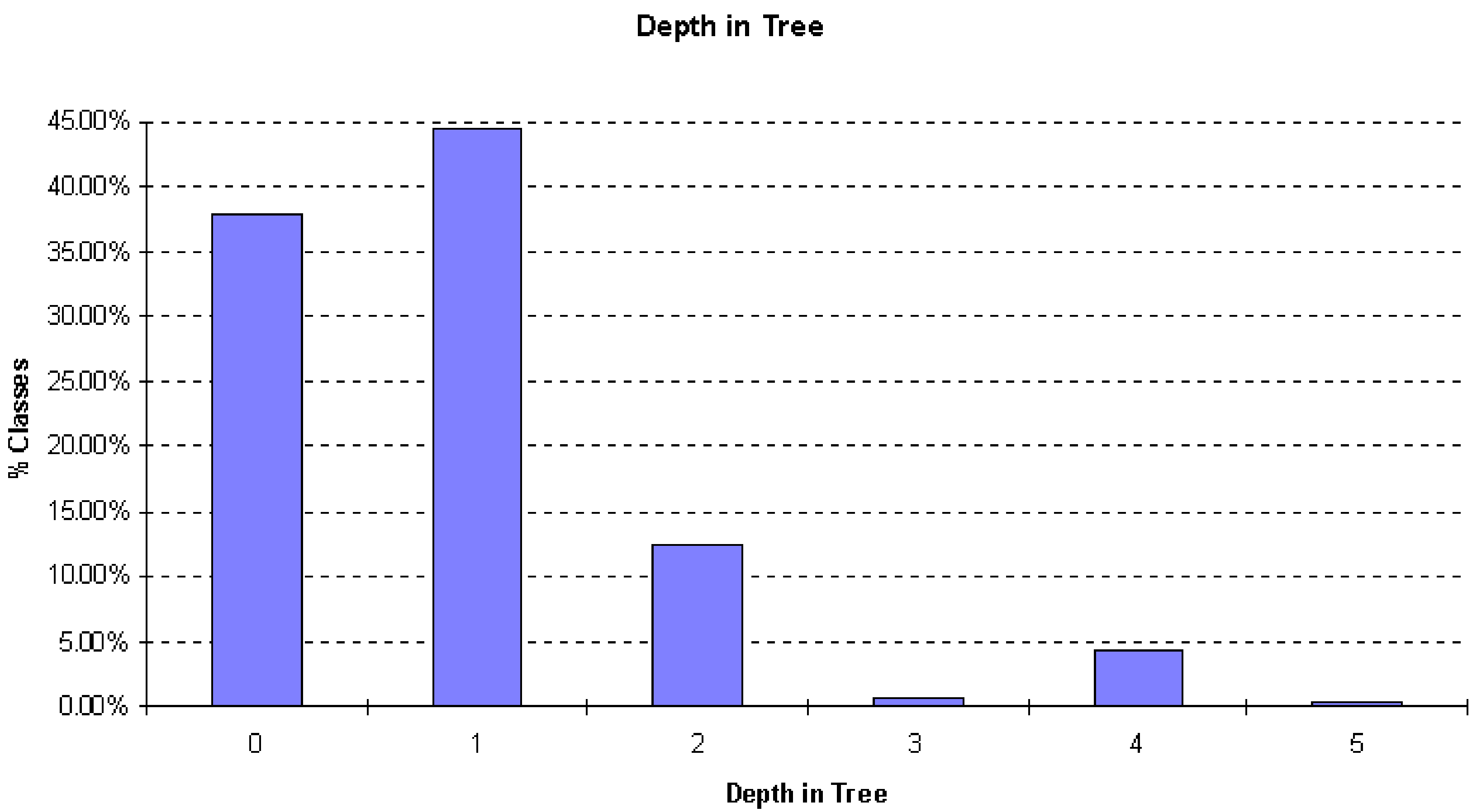
# Number of Children (NOC)

- 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 hierarch 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)



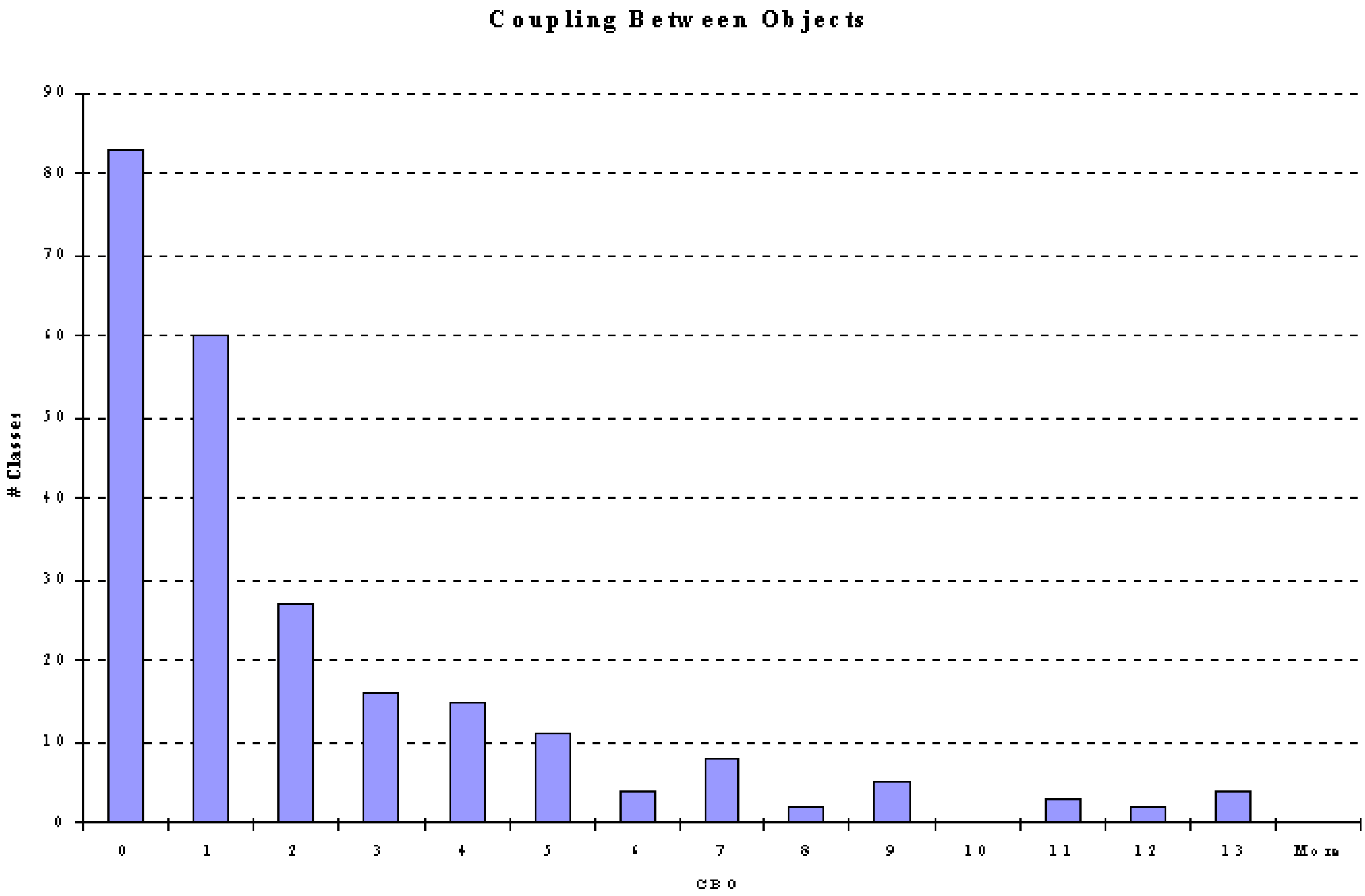
# Coupling Between Objects (CBO)

- 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 → More Testing
- **Rule of thumb:** Low coupling but high cohesion

# Coupling Between Objects (CBO)



# Response for a Class (RFC)

$$RFC = |RS|$$

where RS is the ***response set*** of a class

$$RS = \{M_i\} \cup \{R_i\}$$

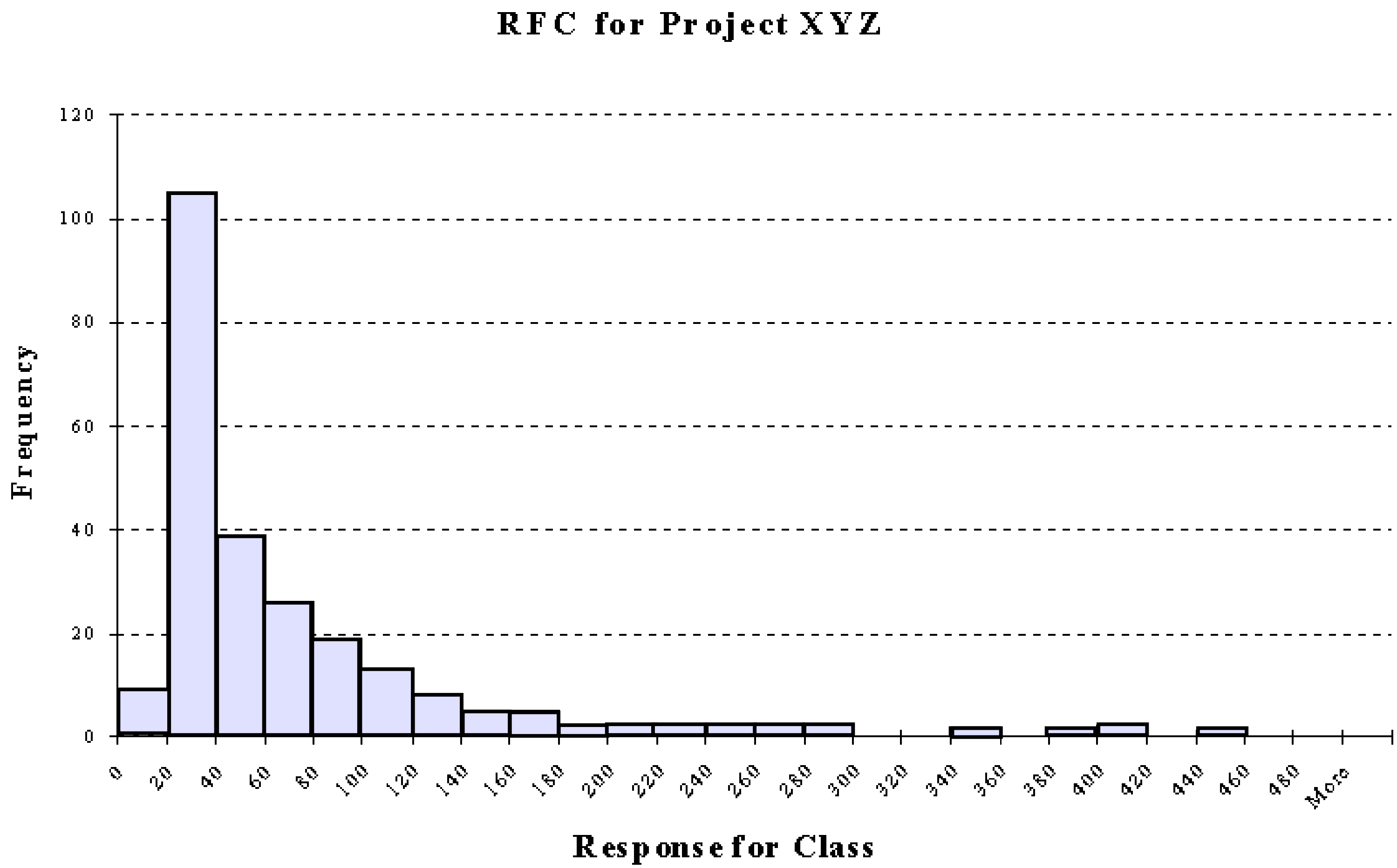
$M_i$ = All the methods in a class

$R_i$ = 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)

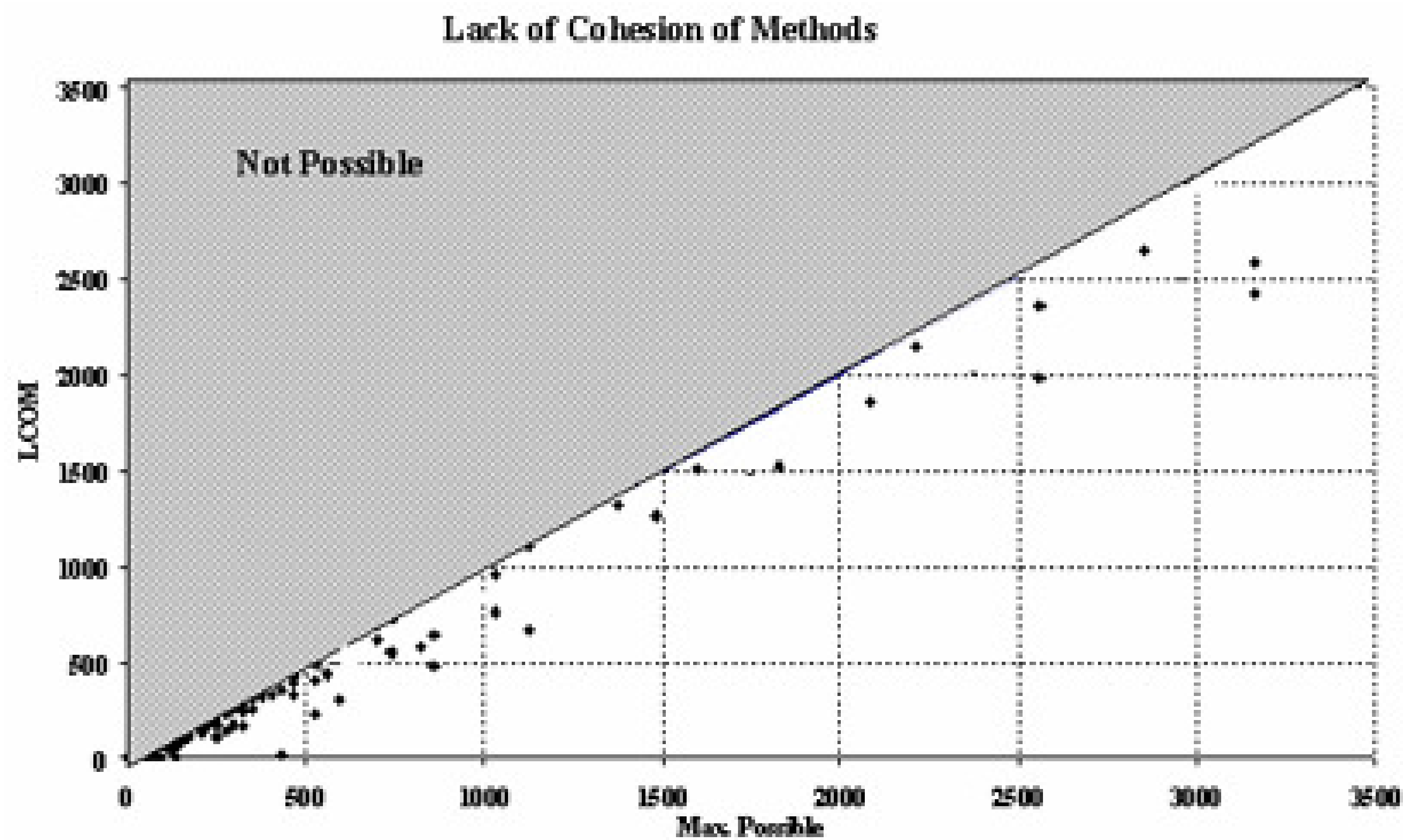
- Consider a class $C_1$ with methods $M_1$, $M_2$, … $M_n$
- Let $\{I_i\}$ be the set of instance variables used by methods $M_i$
- There are n such sets: $\{I_1\}$, $\{I_2\}$, … $\{I_n\}$

LCOM = The number of disjoint sets formed by the intersection of the n 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 ➔ 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