SOFTWARE QUALITY

CPTS 583

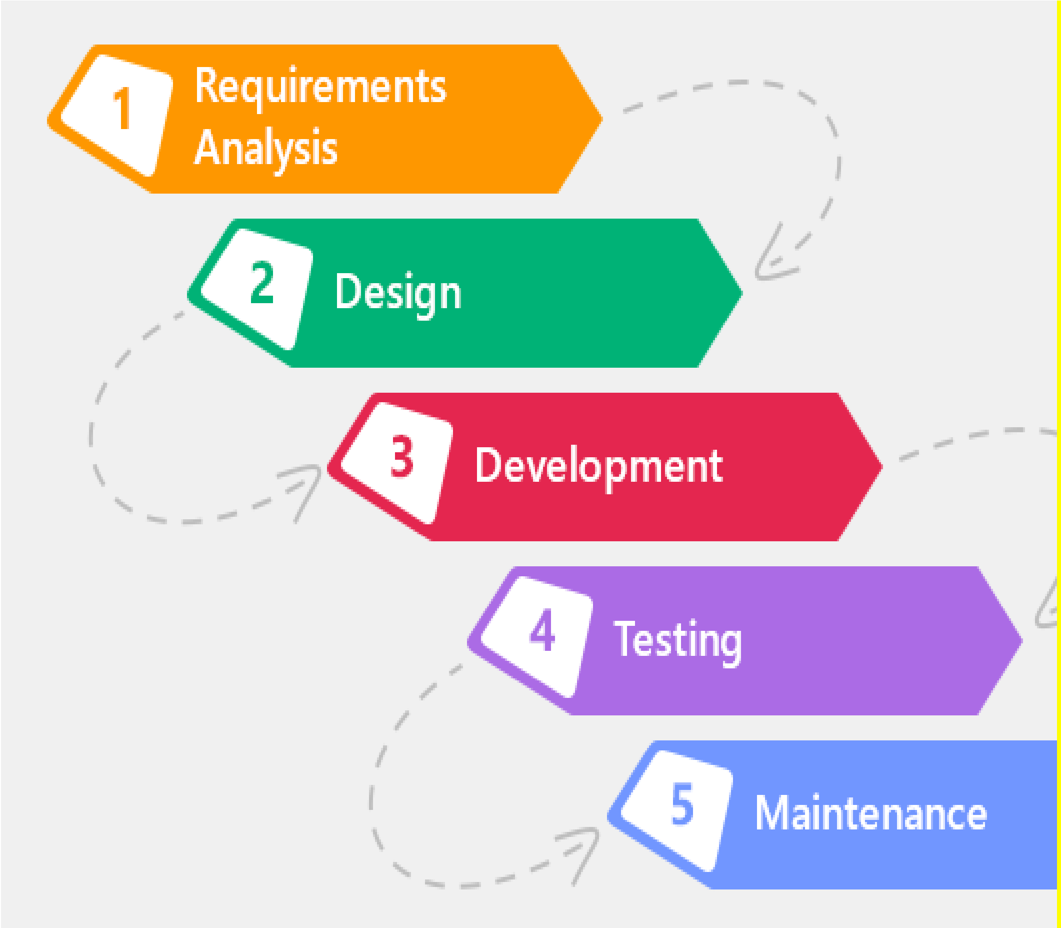
Software Product Quality Metrics and Measurement (III)

-- *Analysis and design metrics*

Outline

- Analysis metrics
  - Size, specificity, volatility of requirements
- General design metrics
  - Architectural design metrics
  - Component-level design metrics
- Object-oriented design metrics
  - Data/Class design metrics
  - Inheritance related metrics
- Other product metrics
  - Testing metric
  - Maintenance metric

Measuring Quality of Work Products



- Requirements model
- Design model
- Code
- Tests
- Bug fixes, etc.

Requirements Analysis Metric

- Size of requirements
  - n_f = number of functional requirements
  - n_{nf} = number of nonfunctional requirements
  - n_r = number of requirements

$$n_r = nf + nnf$$

- Specificity (lack of ambiguity)
  - n_{ui} = number of requirements for which all reviewers had identical interpretations

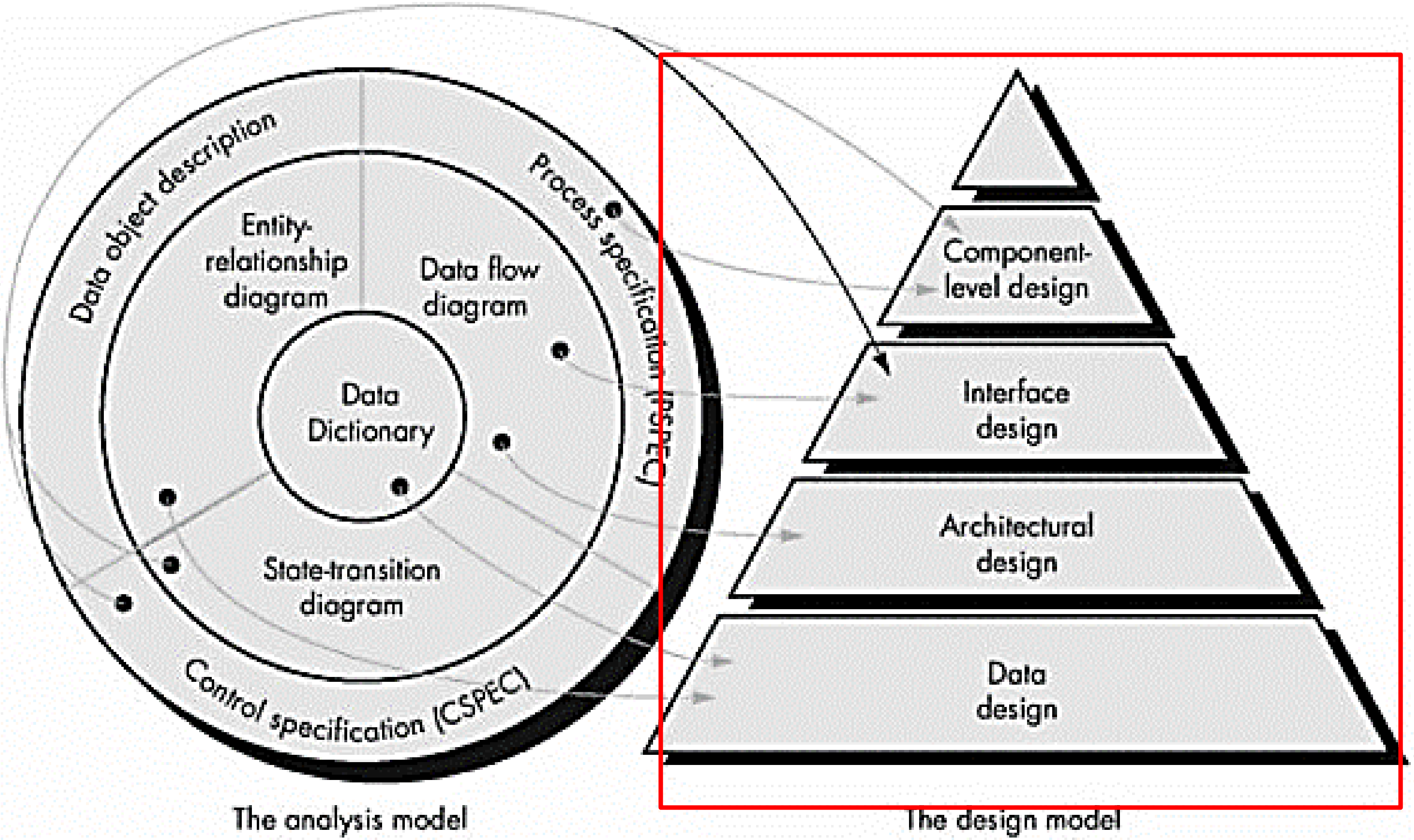
$$Specificity: Q = \frac{n_{ui}}{n_r}$$

Requirements Analysis Metric

- Number of requirements that change during the rest of the software development process
  - if a large number changed during specification, design, …, something is wrong in the requirements phase and the quality of requirements engineers' work

$$Volatility = \frac{\#requirements\ changed}{n_r}$$

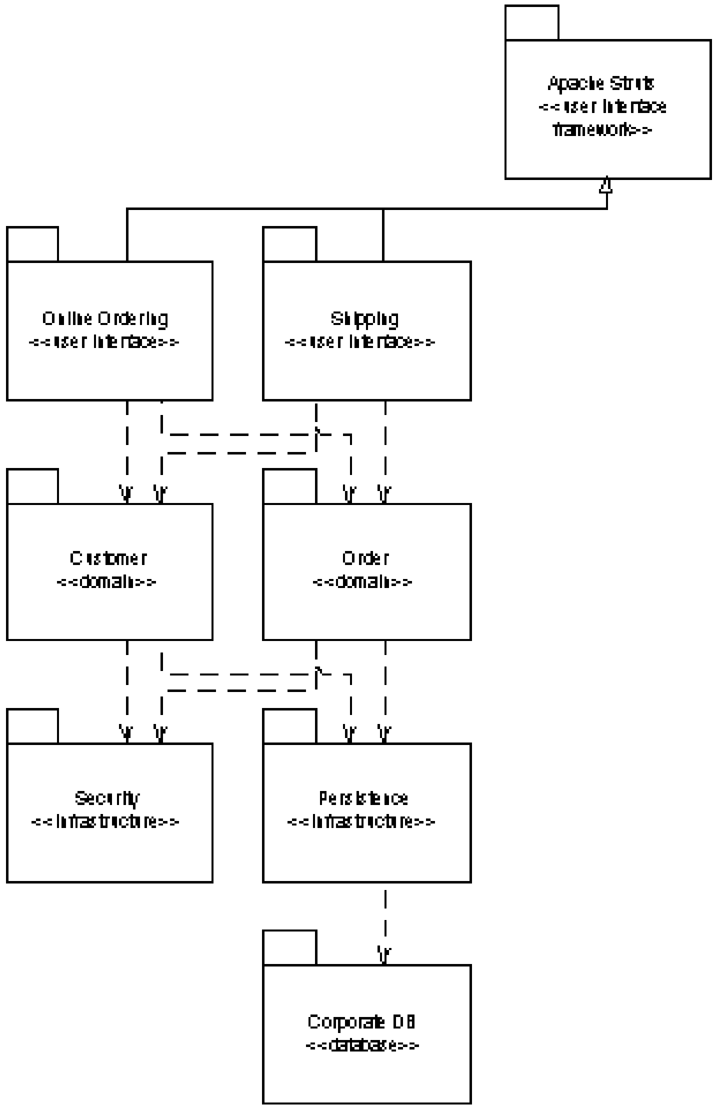
Design Quality Metrics



Mapping from requirements to design model

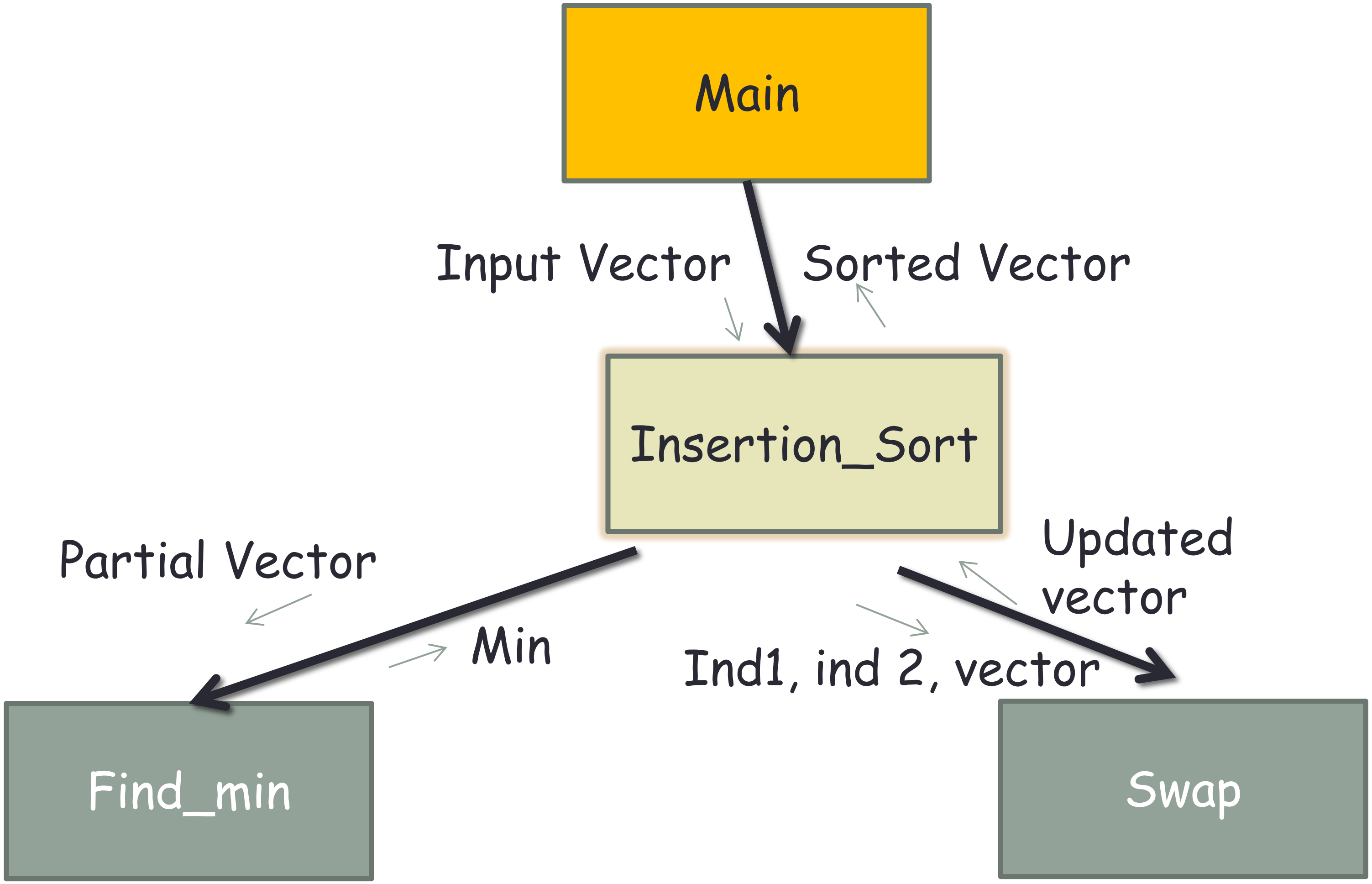
Architectural Design Metrics

- Structural Complexity
  - $S(i) = f^2_{out}(i)$
  - $f_{out}(i)$ = fan-out of module i
- Data Complexity
  - $D(i) = v(i)/[f_{out}(i) + 1]$
  - v(i) = # of input and output variables to and from module i
- System Complexity
  - $C(i) = S(i) + D(i)$



Architectural Design Metrics

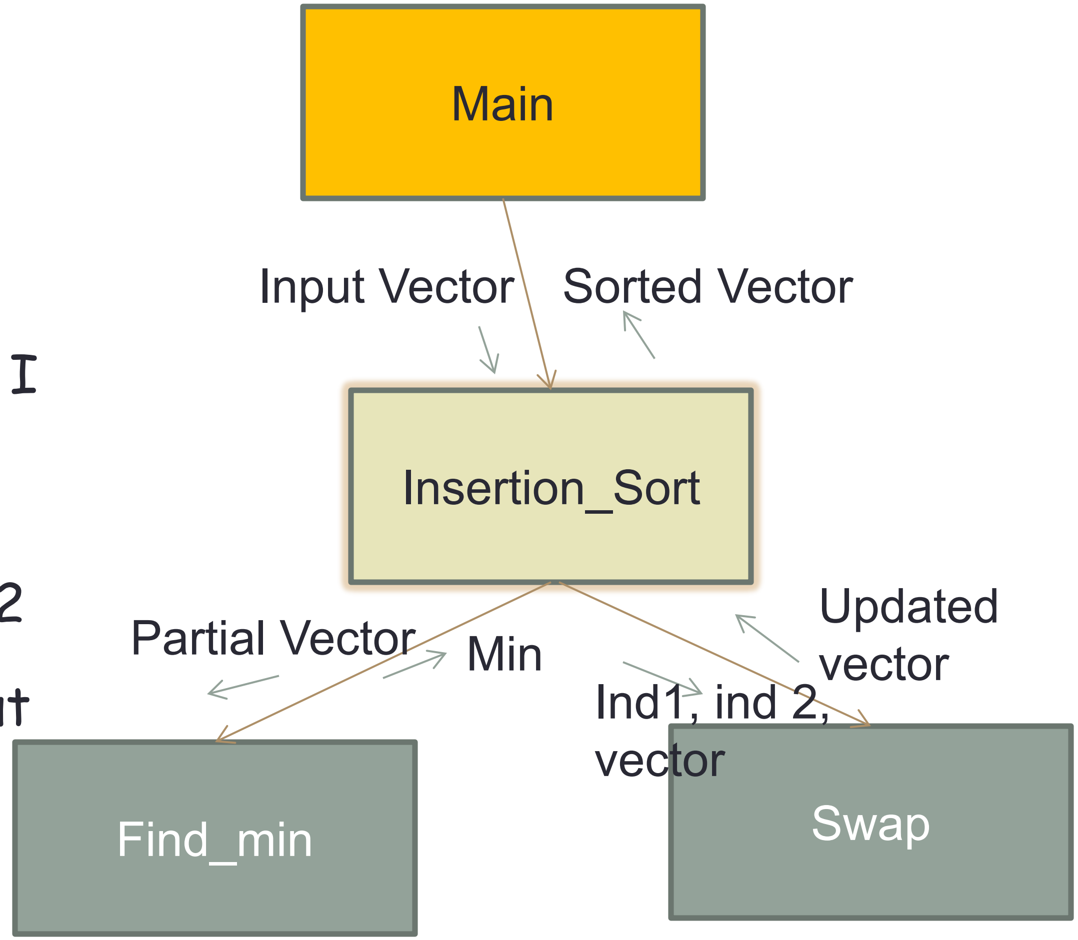
- Example



Architectural Design Metrics

Complexity of Module i:
Insertion_Sort

- Structural Complexity
  - $S(i) = f^2_{out}(i) = 2^2 = 4$
  - $f_{out}(i)$ = fan-out of module I = 2
- Data Complexity
  - $D(i) = v(i)/[f_{out}(i) + 1] = 6/(2 + 1) = 6/3 = 2$
  - v(i) = # of input and output variables to and from module i
- System Complexity
  - $C(i) = S(i) + D(i) = 4 + 2 = 6$

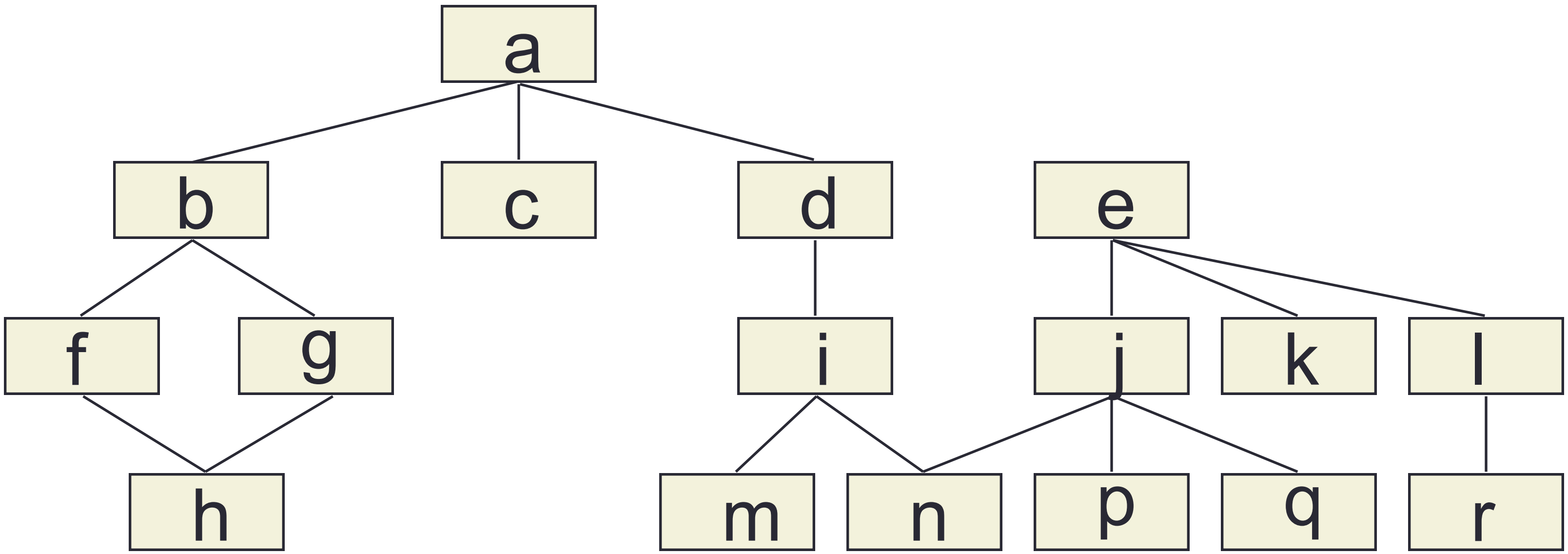


Architectural Design Metrics

- Morphology Metrics
  - size = n + a
  - n = number of modules
  - a = number of arcs (lines of control)
  - arc-to-node ratio, r = a/n
  - depth = longest path from the root to a leaf
  - width = maximum number of nodes at any level

Architectural Design Metrics

- Morphology Metrics – example



size: **17 + 17** arc-to node ratio:**1**
depth:**4** width:**6**

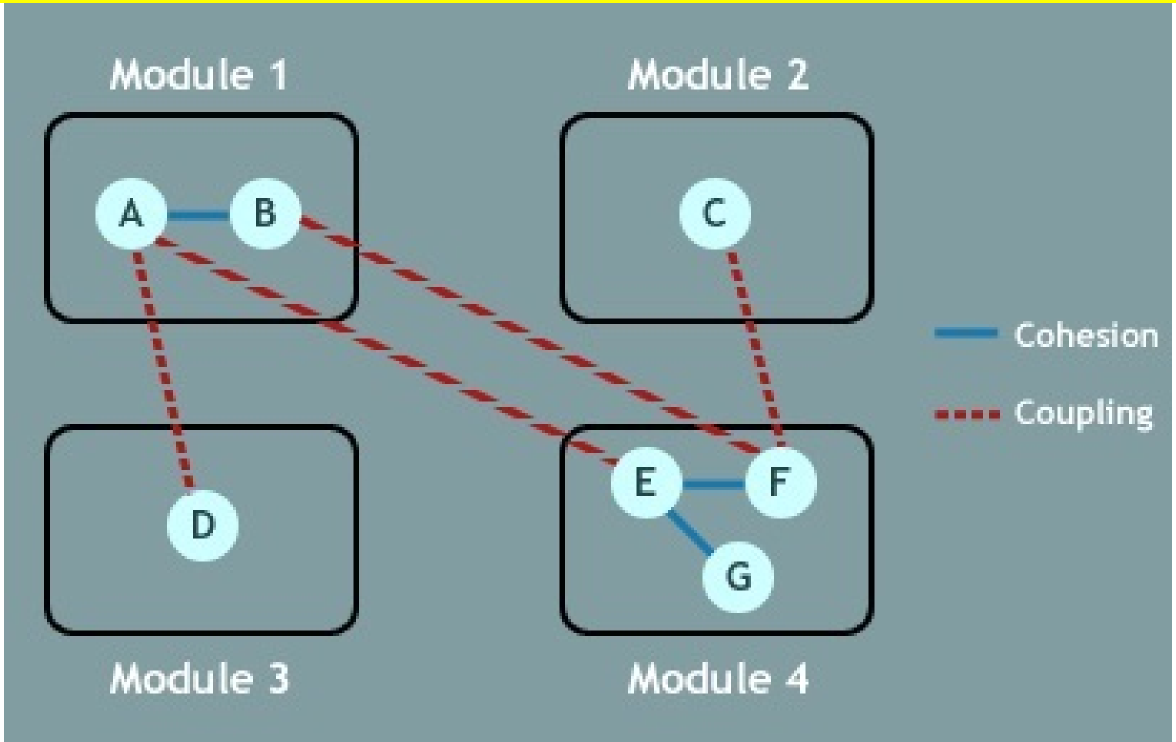
Component-Level Design Metrics

- Cohesion

  the degree to which the elements inside a module belong together

- Coupling

  the degree of interdependence between software modules



Lack of Cohesion in Methods (LCOM)

- Consider a component (e.g., class) C_1 with submodules (e.g., methods) M_1, M_2, … M_n
- Let $\{I_i\}$ be the set of non-static (e.g., instance) variables used by submodule 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)

```
public class GoodApp {
    private double diff, min, max;
    public init(double x){
        this.min = x;
        this.diff = x^2;
    }

    public void output(){
        printf (“%f %f\n”, max, diff);
    }

    public double read() {
        double value;
        do {
            value = ConsoleInput.readDouble();
        } while (value < min || value > max);
        return value;
    }
}
```

- $I_{init} = \{min, diff\}$
- $I_{output} = \{max, diff\}$
- $I_{read} = \{min, max\}$

- Disjoint sets: {diff}, {max}, {min}

- LCOM = 3

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

Coupling Metrics

- Coupling between objects (CBO)
  - Number of other classes to which given class is coupled
  - Interpreted as "number of other classes a class requires to compile"

- Data and control flow coupling
  - di = number of input data parameters
  - do = number of output data parameters

  - ci = number of input control parameters
  - co = number of output control parameters

Coupling Metrics

- Global coupling
  - g_d = number of global variables used as data
  - g_c = number of global variables used as control

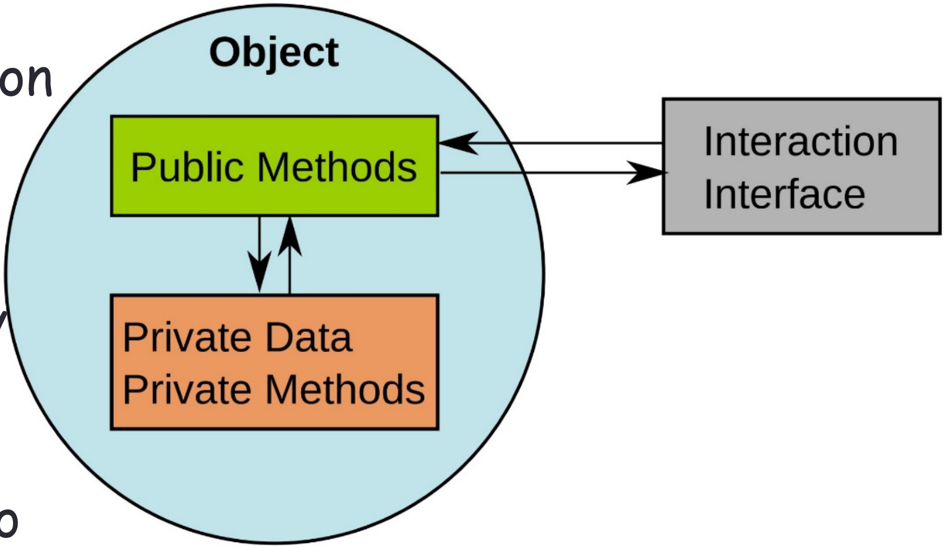
- Environmental coupling
  - w = number of modules called (fan-out)
  - r = number of modules calling the module under consideration (fan-in)

  - Module Coupling: $m_c = 1/ (d_i + 2*c_i + d_0 + 2*c_0 + g_d + 2*g_c + w + r)$

  - $m_c = 1/(1 + 0 + 1 + 0 + 0 + 0 + 1 + 0) = .33$ (Low Coupling)
  - $m_c = 1/(5 + 2*5 + 5 + 2*5 + 10 + 0 + 3 + 4) = .02$ (High Coupling)

Object Oriented Design Metrics

- Special metrics needed due to OO characteristics
  - Encapsulation — Concentrate on classes rather than functions
  - Information hiding — An information hiding metric will provide an indication of quality
  - Inheritance — A pivotal indication of complexity
  - Abstraction — Metrics need to measure a class at different levels of abstraction and from different viewpoints
- the class is the fundamental unit of measurement



Characteristics of Object Orientation

- **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

Characteristics of Object Orientation

- **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

Measurement Structures in OO

- **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

Measurement Structures in OO

- **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

Measurement Structures in OO

- **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)

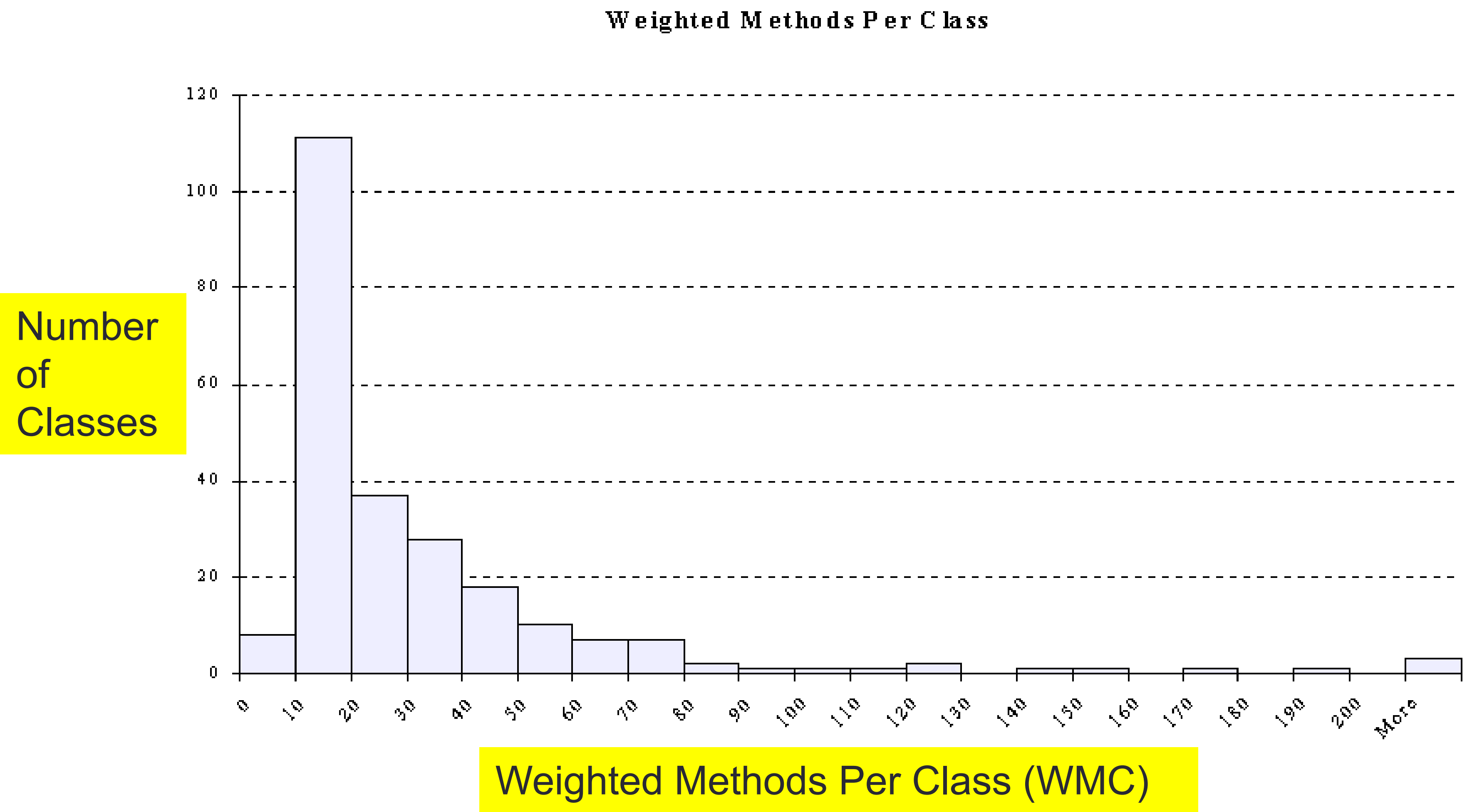
- Consider the class C with methods m_1, m_2, … $m_{n.}$
- Let c_1, c_2 … c_n be the complexity of these methods.

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

- Complexity of objects of this class
- Time and effort required for development
- Complexity inheritance
- Larger class more application-specific and less reusable
- WMC of [20,40] as a good guideline

Weighted Methods Per Class (WMC)

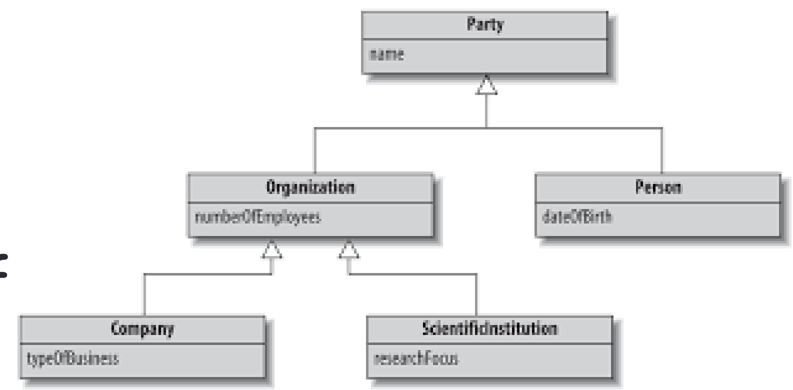
- WMC in an industry project



Number of Children (NOC)

- Count immediate subclasses of a particular class

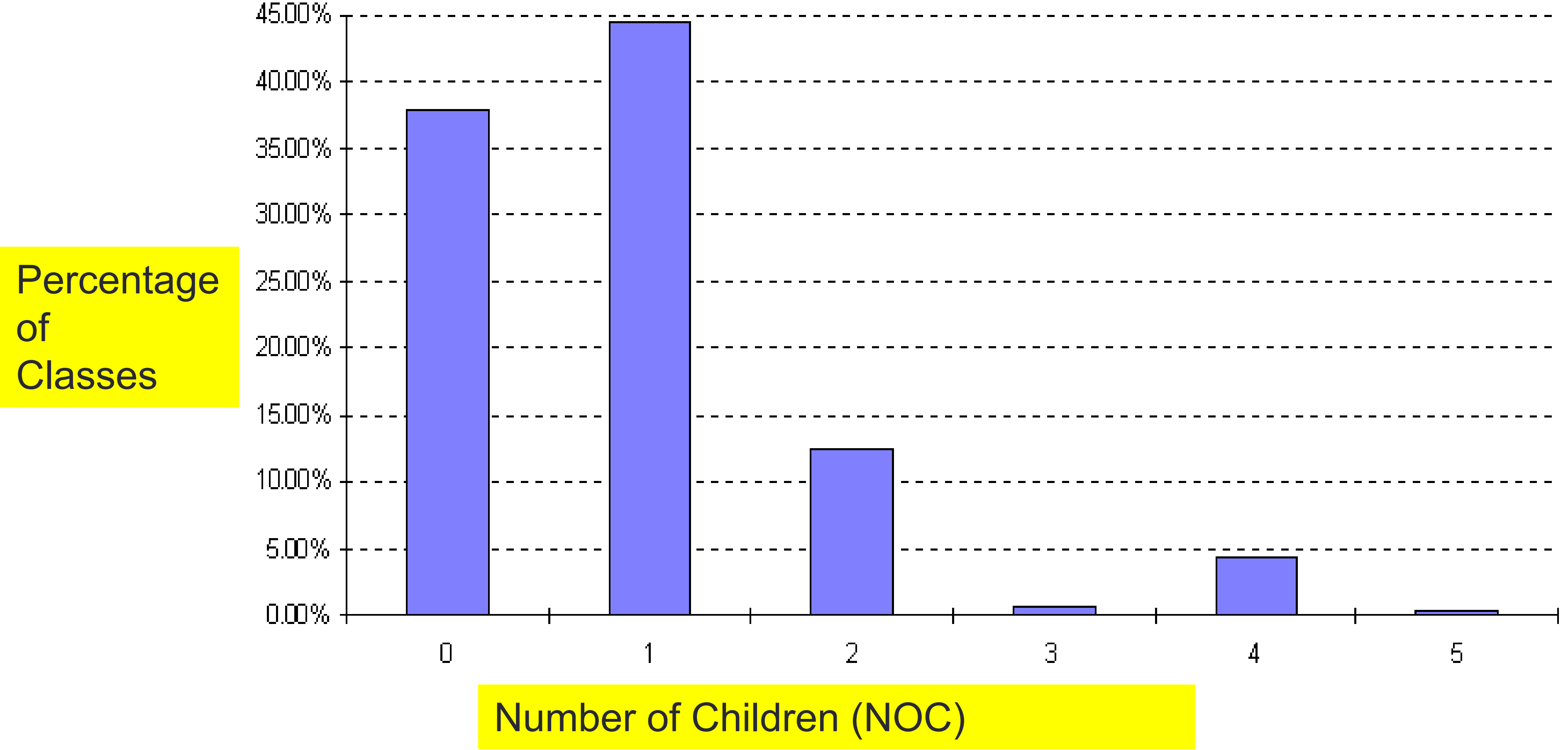
- Prefer depth over breadth: better reusability
- Classes higher up in the hierarch have larger NOC



- Indicates: potential influence of a class on design
- Affects: Efficiency, Reusability, Testability

Number of Children (NOC)

- An industrial case of NOC



Response for a Class (RFC)

$$RFC = |RS|$$

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

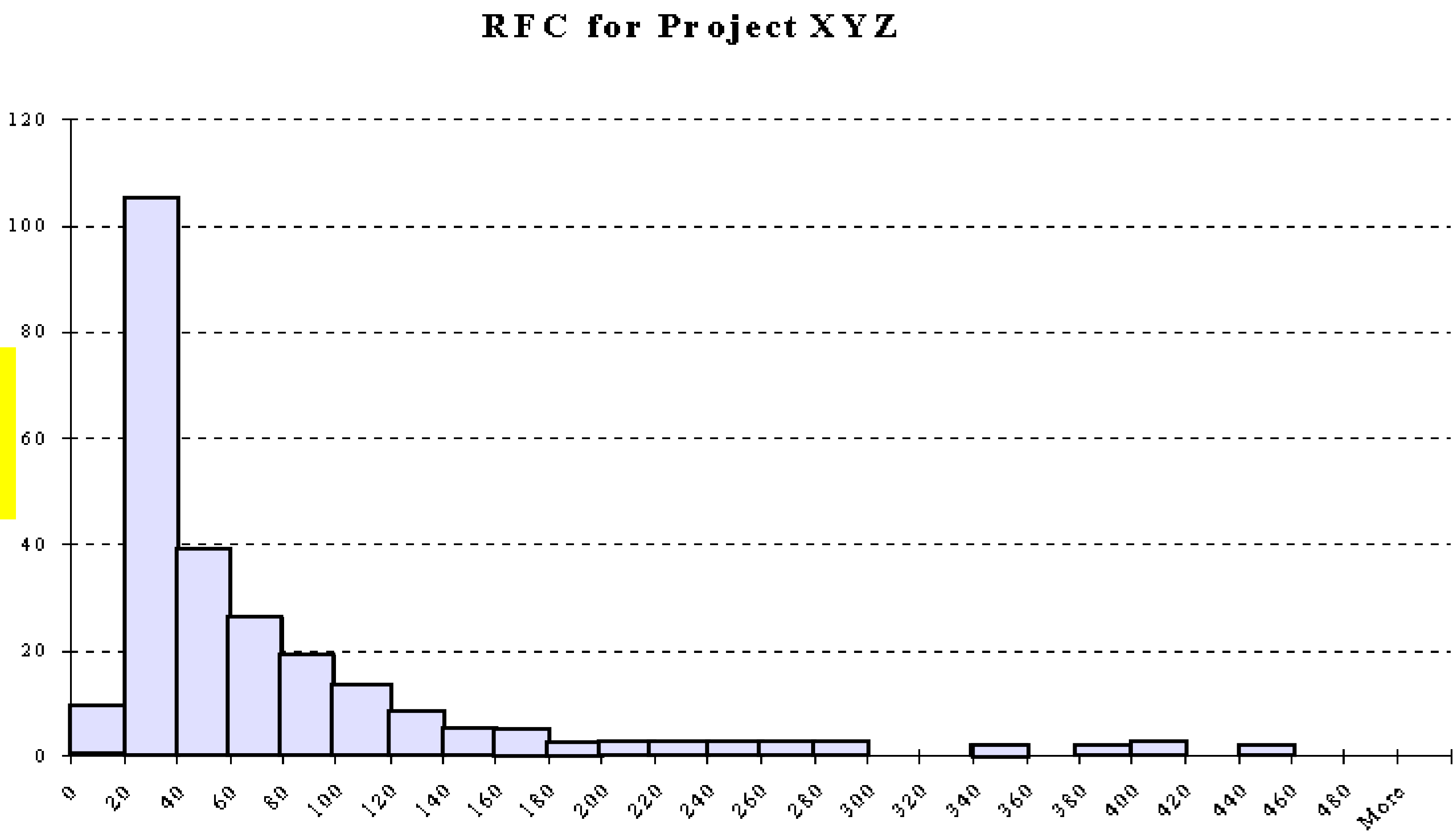
$$RS = \{M\} \cup \{R\}$$

M_i = All the methods in a class

R_i = All methods called by that class

- More methods invoked → More Complex Object
- Affects: understandability, maintainability, testability

Response for a Class (RFC)



Number of Classes

Response for a Class (RFC)

Response for a Class (RFC)

- Computing RFC for the following case

RFC = 7

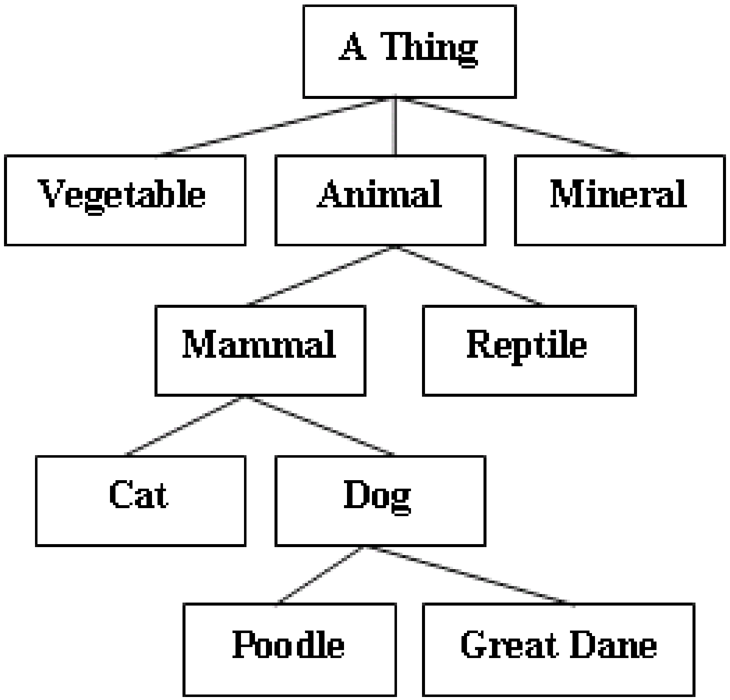
new PageSecurityService(…).hasAccessTo (…)

```
class PageSecurityService {
  PageSecurityService(SecurityContext securityContext) { ... }

  boolean hasAccessTo(User user, Page page) {
    return !securityContext.getGlobalLock().isEnabled() &&
           securityContext.getApplicationContext()
           .getSecurityDao().userHasPermission(user, page);
  }
}
```

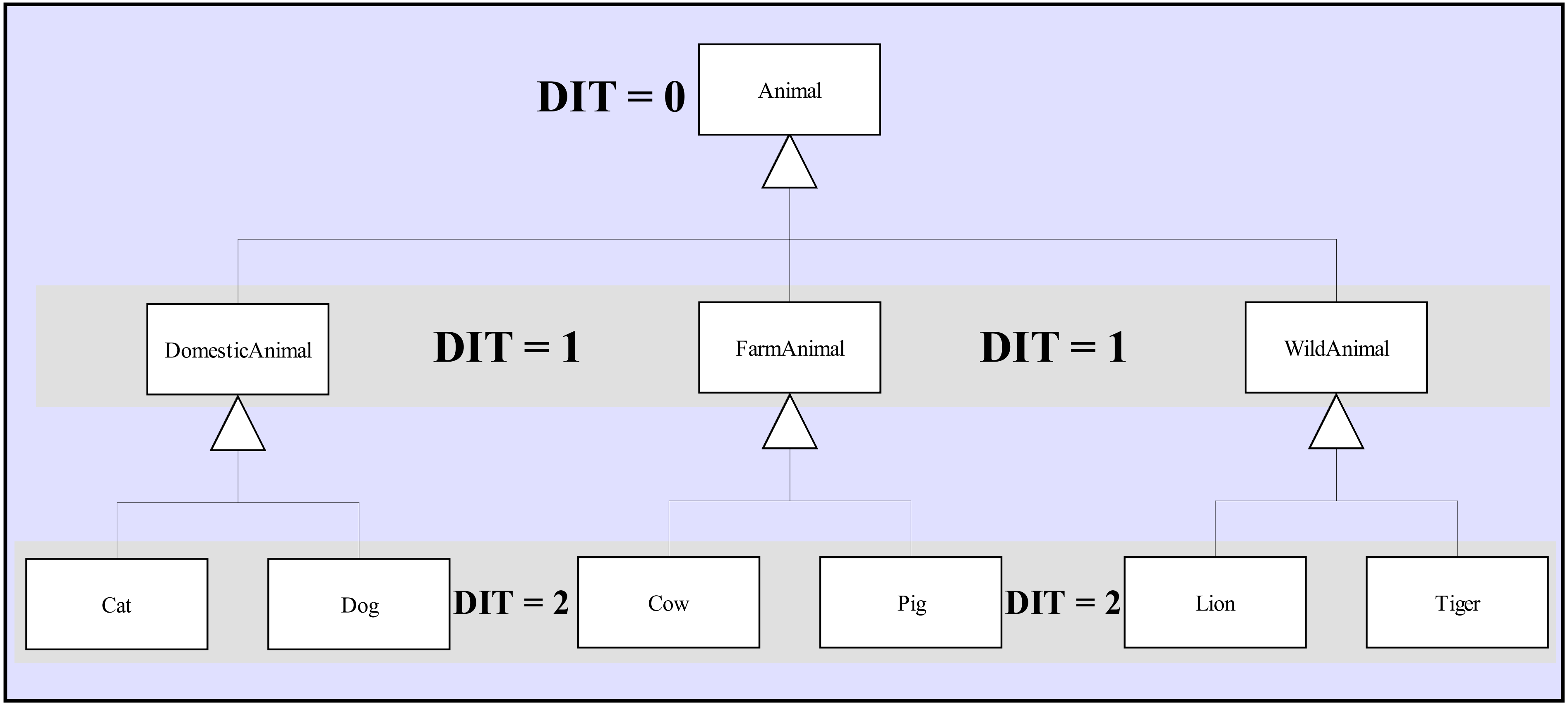
Depth of Inheritance Tree (DIT)

- 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



Depth of Inheritance Tree (DIT)

- Quick illustration



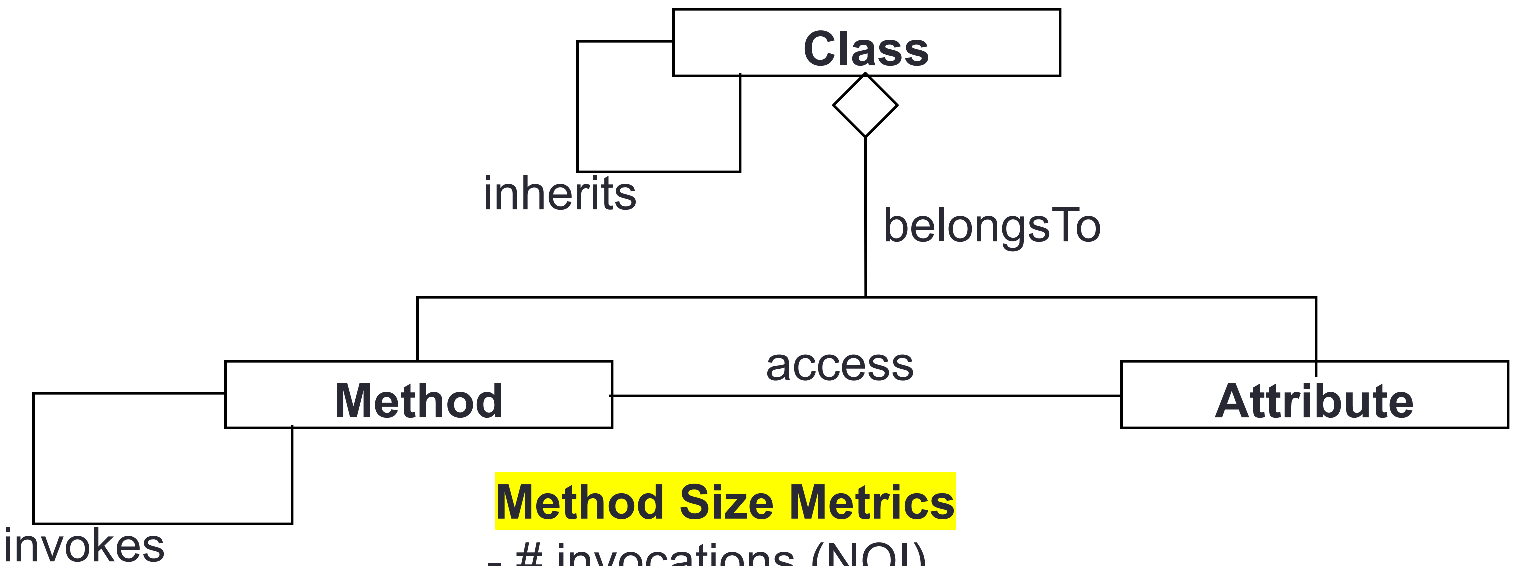
Depth of Inheritance Tree (DIT)

- The deeper a class is in the hierarchy, the greater the number of methods likely to inherit from parent classes – ***more complex***
- Deeper trees → Greater Design Complexity
- Deeper trees → More Reuse
- Affects: efficiency, reuse, understandability and testability

More OO design metrics

Inheritance Metrics
- hierarchy nesting level (HNL)
- # inherited methods, unmodified (NMI)
- #overridden methods (NMO)

Class Size Metrics
- # methods (NOM)
- # attributes, instance/class (NIA, NCA)



Method Size Metrics
- # invocations (NOI)
- # statements (NOS)
- # arguments (NOA)

More OO analysis and design metrics

- Number of Scenario Scripts (Use Cases):
  - Number of use-cases is directly proportional the number of classes needed to meet requirements
  - A strong indicator of program size
- Number of Key Classes (Class Diagram):
  - A key class focuses directly on the problem domain
  - NOT likely to be implemented via reuse
  - Typically 20-40% of all classes are key, the rest support infrastructure (e.g. GUI, communications, databases)
- Number of Subsystems (Package Diagram):
  - Provides insight into resource allocation, scheduling for parallel development and overall integration effort

Testing metric

- Test cases as the targeted product
- Guide the design and execution of test cases

- Measuring testing completeness
  - Breadth of Testing

total number of requirements covered by testing

  - Depth of Testing

$$\frac{\text{\# independent basis paths covered by testing}}{\text{\# independent basis paths in the program.}}$$

Cyclomatic complexity

Testing metric

- Number of test cases executed
- Number of bugs found per thousand of code

- Inspection effectiveness
  - number of errors found during inspection can be used as a metric to the quality of inspection process

Maintenance metric

- Software Maturity Index (SMI)

$$SMI = [M_T - (F_c + F_a + F_d)] / M_T$$

- M_T = number of modules in the current release
- F_c = number of modules in the current release that have been changed
- F_a = number of modules in the current release that have been added
- F_d = number of modules from the preceding release that were deleted in the current release

Maintenance metric

- Total number of faults reported
- Classifications by severity, fault type
- Status of fault reports (reported/fixed)
- Detection and correction times

Summary

- Requirements size, specificity, and volatility
  - More: number of use cases/scenarios
- Design metrics
  - General design: architectural (structure, data, system complexity), component level (LCOM, CBO, global coupling, environmental coupling, data/control flow coupling)
  - OO design: data/class design (WMC, NOC, RFC), inheritance metrics (DIT)
  - More: number of key classes, number of subsystems
- Testing metrics
  - Number of test cases, number of bugs found per KLOC
- Maintenance metrics
  - Software maturity index (SMI)