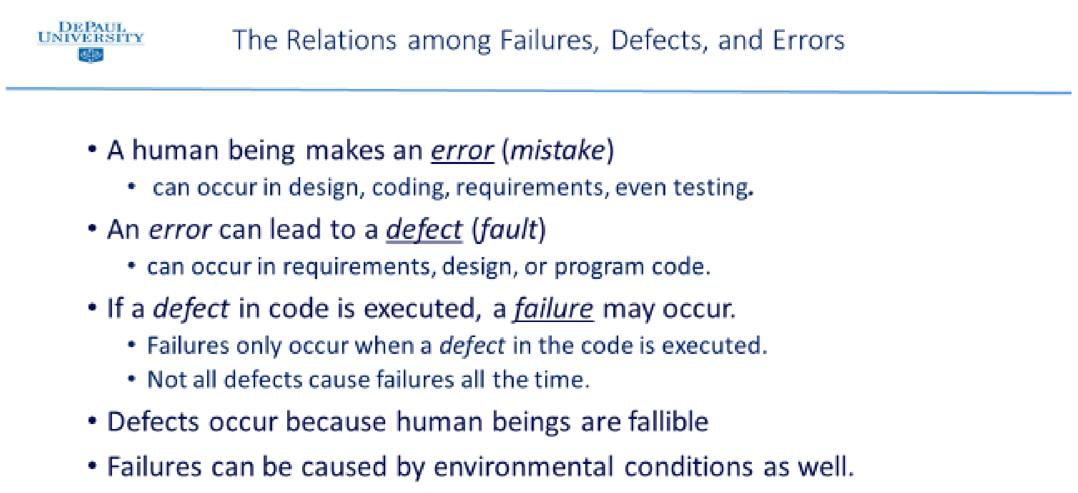
DEPAUL UNIVERSITY

SE433 Software testing & Quality assurance

Software Quality Metrics

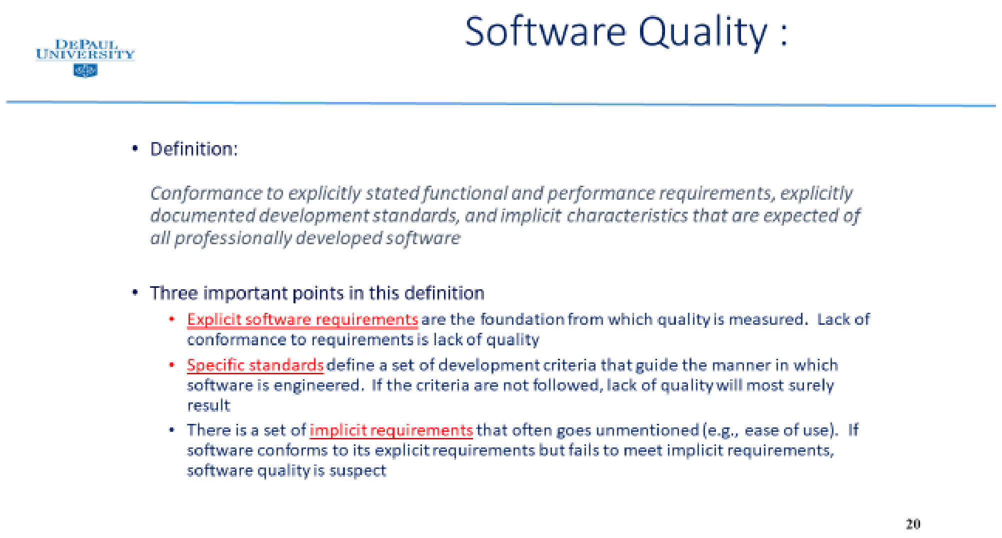
DEPAUL UNIVERSITY

Last Week



The Relations among Failures, Defects, and Errors

- A human being makes an *error* (*mistake*)
  - can occur in design, coding, requirements, even testing.
- An *error* can lead to a *defect* (*fault*)
  - can occur in requirements, design, or program code.
- If a *defect* in code is executed, a *failure* may occur.
  - Failures only occur when a *defect* in the code is executed.
  - Not all defects cause failures all the time.
- Defects occur because human beings are fallible
- Failures can be caused by environmental conditions as well.



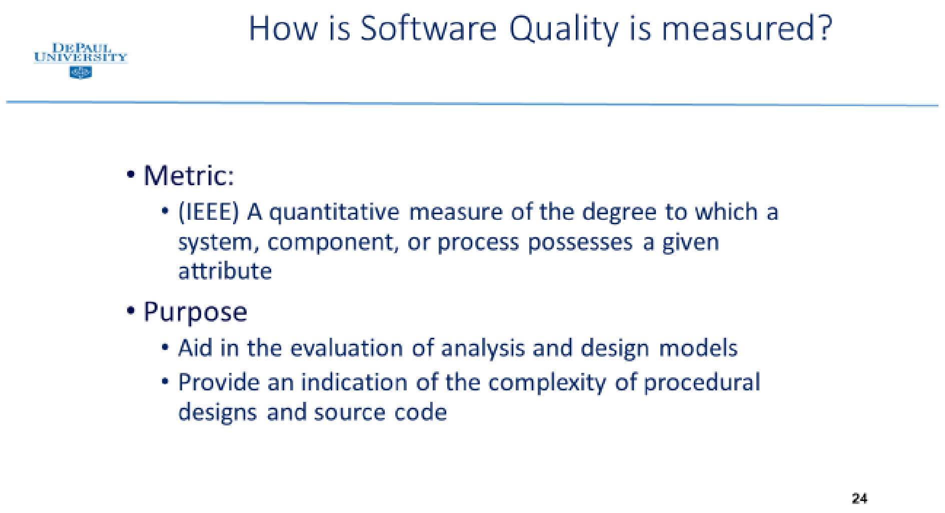
Software Quality :

- Definition:

  Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software

- Three important points in this definition
  - Explicit software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality
  - Specific standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will most surely result
  - There is a set of implicit requirements that often goes unmentioned (e.g., ease of use). If software conforms to its explicit requirements but fails to meet implicit requirements, software quality is suspect

20



How is Software Quality is measured?

- Metric:
  - (IEEE) A quantitative measure of the degree to which a system, component, or process possesses a given attribute
- Purpose
  - Aid in the evaluation of analysis and design models
  - Provide an indication of the complexity of procedural designs and source code

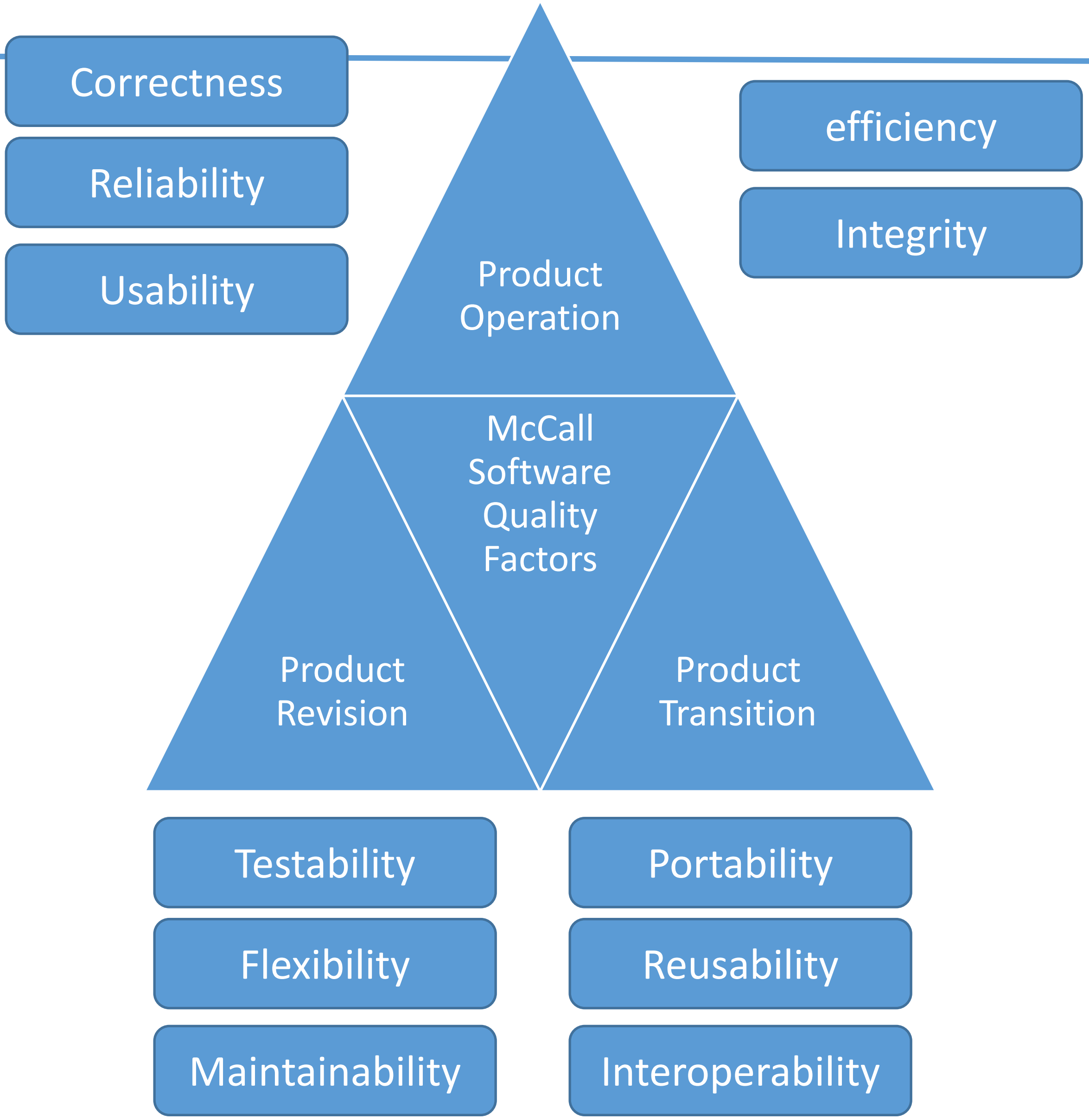
24

- Software quality model
  - Quality factor
  - Quality criteria
  - Quality metrics
    - Software metrics: overview
    - Metrics for various phases
    - Why metrics are needed
      - How to collect metrics
      - How to use metrics

McCall Model Elements

- Jim McCall produced this model for the US Air force
- McCall quality model is organized around three types of elements
  - Quality factors : they describe the external view of the software, as viewed by the users
  - Quality criteria : they describe the internal view of the software, as seen by the developer
  - Quality Metrics : they are defined and used to provide a scale and method for measurement

McCall's quality factors



McCall’s quality factors: Product Operation

1. Correctness: Extent to which a program satisfies its specifications and fulfills the user’s mission objectives. It can be calculated as

= (No. of requirements fulfilled) / (Total no. of requirements) * 100

2. Reliability: Extent to which a program can be expected to perform its intended function with required precision. The formula is

= (Mean Time To Failure) / (Total Run Time) * 100 or = (Mean Time Between Failure) / (Total Run Time) * 100

3. Efficiency: The amount of computing resources and code required by a program to perform a function.

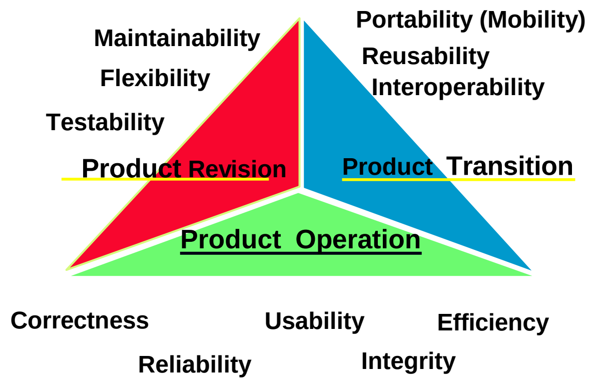
= (Memory Usage) / (Total Memory) * 100

4. Integrity: Extent to which access to software or data by unauthorized persons can be controlled.

= (No. of successful attempts) / (Total no. of attempts) * 100

5. Usability: Effort required learning, operating, preparing input, and interpreting output of a program.

= (Total Training Time) / (Total development time) * 100

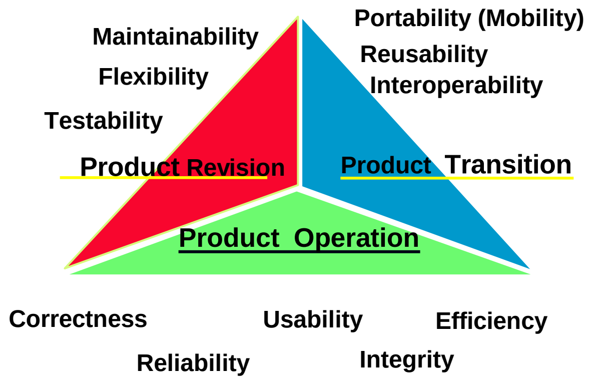


McCall's quality factors : Product Revision

6. Maintainability: Effort required locating and fixing an error in an operational program.
= (Time spent to fix a bug) / (Total development time) * 100

7. Testability: Effort required testing a program to ensure that it performs its intended function.
= (Time spent in testing the functionality) / (Total development time) * 100

8. Flexibility: Effort required modifying an operational program.
= (Time spent to apply changes and additions to the software) / (Total development time) * 100

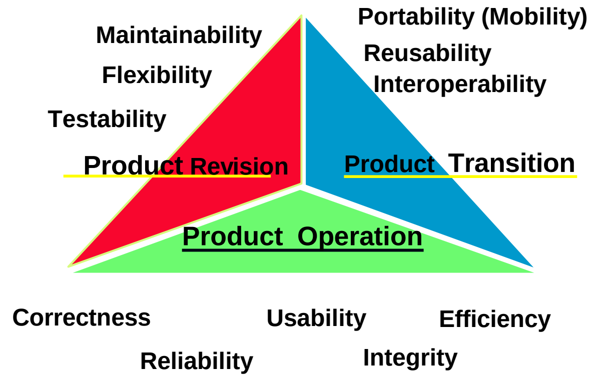


McCall's quality factors: Product Transition

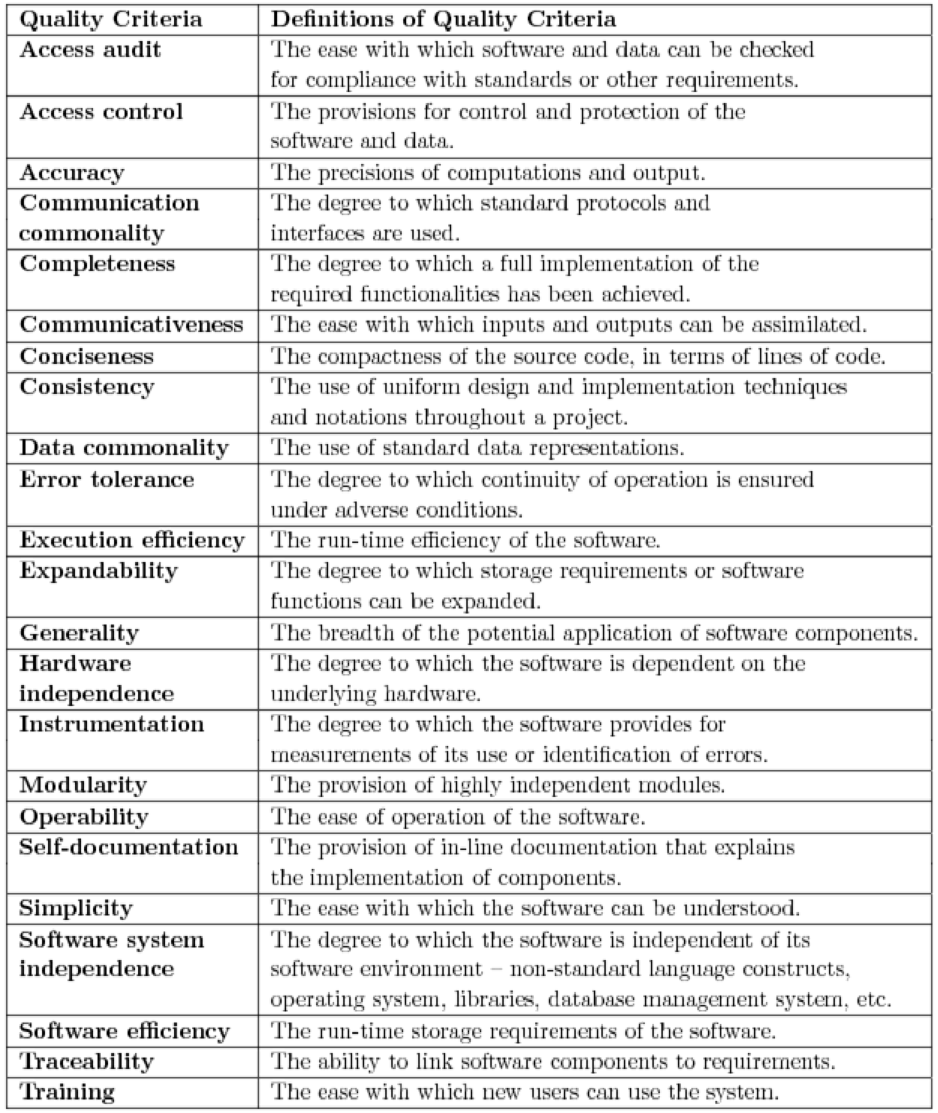
9. Portability: Ability to reuse the existing code instead of creating new code when moving software from an environment to another.
= (No. of successful ports) / (Total no. of ports) * 100

10. Reusability: Ability of program to be used in other applications. It is related to the program packaging and scope of its functions
= (No. of reusable components) / (Total no. of components) * 100

11. Interoperability: Effort required to couple one system with another.
= (Time spent in coupling the system) / (Installation Time) * 100



McCall’s quality criteria



Quality Criteria	Definitions of Quality Criteria
Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
Access control	The provisions for control and protection of the software and data.
Accuracy	The precisions of computations and output.
Communication commonality	The degree to which standard protocols and interfaces are used.
Completeness	The degree to which a full implementation of the required functionalities has been achieved.
Communicativeness	The ease with which inputs and outputs can be assimilated.
Conciseness	The compactness of the source code, in terms of lines of code.
Consistency	The use of uniform design and implementation techniques and notations throughout a project.
Data commonality	The use of standard data representations.
Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
Execution efficiency	The run-time efficiency of the software.
Expandability	The degree to which storage requirements or software functions can be expanded.
Generality	The breadth of the potential application of software components.
Hardware independence	The degree to which the software is dependent on the underlying hardware.
Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.
Modularity	The provision of highly independent modules.
Operability	The ease of operation of the software.
Self-documentation	The provision of in-line documentation that explains the implementation of components.
Simplicity	The ease with which the software can be understood.
Software system independence	The degree to which the software is independent of its software environment – non-standard language constructs, operating system, libraries, database management system, etc.
Software efficiency	The run-time storage requirements of the software.
Traceability	The ability to link software components to requirements.
Training	The ease with which new users can use the system.

Quality Criteria

- A quality criterion is an attribute of a quality factor that is related to software development.

Example: A highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system.

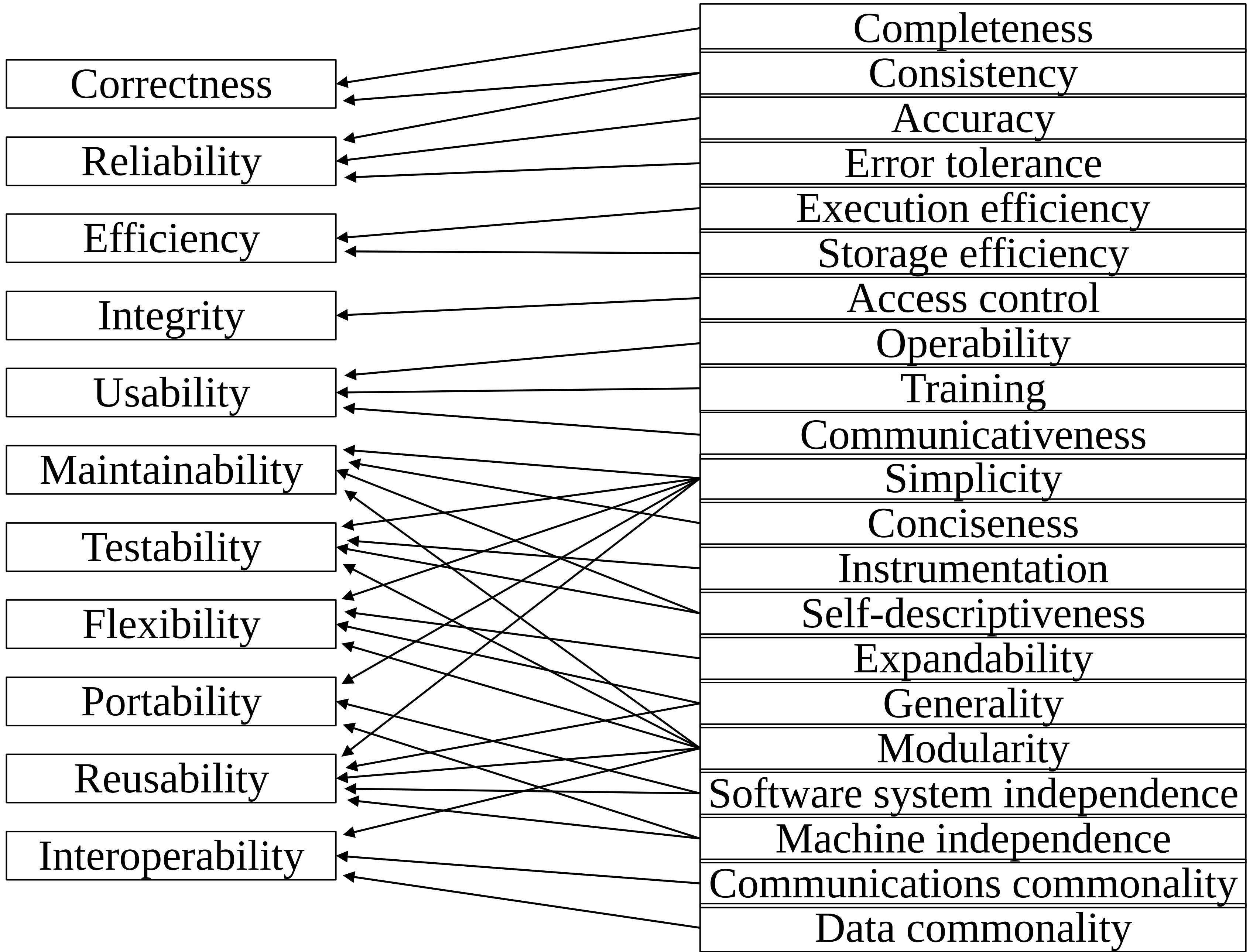
Relationship Between Quality Factors and Quality Criteria

- Each quality factor is positively influenced by a set of quality criteria, and the same quality criterion impacts a number of quality factors.

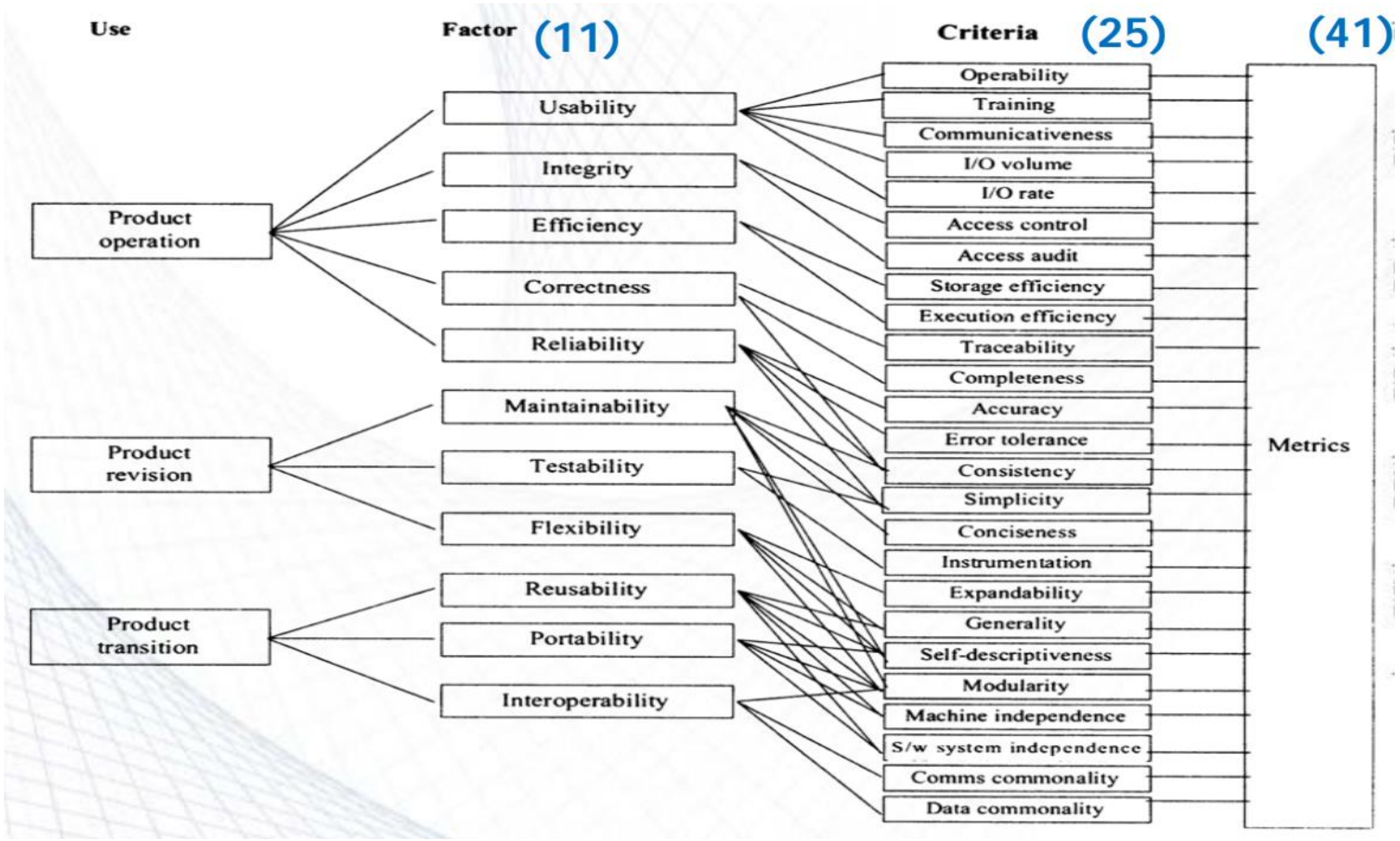
Example: Simplicity impacts usability, and testability.

- If an effort is made to improve one quality factor, another quality factor may be degraded.

Quality Factors with Quality Criteria



McCall’s Model



Learning objectives

- Software metrics: overview
- Metrics for various phases
- Why metrics are needed
    - How to collect metrics
    - How to use metrics

Questions

- How big is the program?
  - Huge!!
- How close are you to finishing?
  - We are almost there!!
- Can you, as a manager, make any useful decisions from such subjective information?
  - Need information like, cost, effort, size of project.

Why Measure Software?

Estimate cost and effort	measure correlation between specifications and final product
Improve productivity	measure value and cost of software
Improve software quality	measure usability, efficiency, maintainability ...
Improve reliability	measure mean time to failure, etc.
Evaluate methods and tools	measure, quality, reliability ...

"You cannot control what you cannot measure" — De Marco, 1982
"What is not measurable, make measurable" — Galileo

What are Software Metrics?

Software metrics

- Any type of measurement which relates to a software system, process or related documentation
  - Lines of code in a program
  - number of person-days required to implement a use-case

Direct and Indirect Measures/Metrics

Direct Measures

- Measured directly in terms of the observed attribute (usually by counting)
- Length of source-code, Duration of process, Number of defects discovered

Indirect Measures

- Calculated from other direct and indirect measures
- Module Defect Density = Number of defects discovered / Length of source

Metrics Suites Object Oriented Design

- Chidamber and Kemerer’s suite (CK - 1994)*
- Brito and Abreu’s suite (MOOD – 1995)
- Bansiya and Davis’s suite (QMOOD– 2002)*

Object-orientation/Terminology

- Class
  - A Class is a description of a set of objects that share the same attributes, operations, relationships, and semantics.
- Object
  - An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.
- Attribute Variable
  - An attribute is a named property of a class that describes a range of values instances of the property may hold
- Operation Responsibility Method
  - An operation is the implementation of a service that van be requested from any object of the class to affect behavior.

Object-orientation/Terminology

- Package
  - A general purpose mechanism for organizing elements into groups. Packages group functionally related classes/
- Cohesion
  - The degree to which the methods within a class or classes in a package are related to one another
- Coupling
  - Object X is coupled to object Y if and only if X sends a message to Y
- Association
  - A semantic relationship between two or more classes that specifies connections among their instances
- Inheritance
  - A relationship among classes, wherein an object in a class acquires characteristics from one or more other classes.

Object

- An object can be considered a “thing” that can perform a set of related activities.
- Set of activities defines the object’s behavior
- Example :
  - Hand (object) can grip something
  - Student can give : name or address
- Object = instance of a class

DePaul University

Class

- A class is a representation of a type of object.
- A class is the blueprint from which the individual objects are created

```
public class Student
{ }

Student objectStudent = new Student();
```

• Inheritance

- Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.
- The keyword used for inheritance is **extends**

```
class derived-class extends base-class
{

//methods and fields

}
```

Inheritance Example

```java
//Java program to illustrate the
// concept of inheritance

// base class
class Bicycle
{
    // the Bicycle class has two fields
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int gear, int speed)
    {
        this.gear = gear;
        this.speed = speed;
    }

    // the Bicycle class has three methods
    public void applyBrake(int decrement)
    {
        speed -= decrement;
    }

    public void speedUp(int increment)
    {
        speed += increment;
    }

    // toString() method to print info of Bicycle
    public String toString()
    {
        return("No of gears are "+gear
                +"\n"
                + "speed of bicycle is "+speed);
    }
}
```

```java
// derived class
class MountainBike extends Bicycle
{

    // the MountainBike subclass adds one more field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int gear,int speed,
                        int startHeight)
    {
        // invoking base-class(Bicycle) constructor
        super(gear, speed);
        seatHeight = startHeight;
    }

    // the MountainBike subclass adds one more method
    public void setHeight(int newValue)
    {
        seatHeight = newValue;
    }

    // overriding toString() method
    // of Bicycle to print more info
    @Override
    public String toString()
    {
        return (super.toString()+
                "\nseat height is "+seatHeight);
    }

}

// driver class
public class Test
{
    public static void main(String args[])
    {

        MountainBike mb = new MountainBike(3, 100, 25);
        System.out.println(mb.toString());

    }
}
```

DEPAUL UNIVERSITY

- **Polymorphism**
  - Provides the ability of a function or a method to have multiple implementations with the same name. The implementation is chosen based on the type of parameters for a function and the type of the object (and parameters) for a method

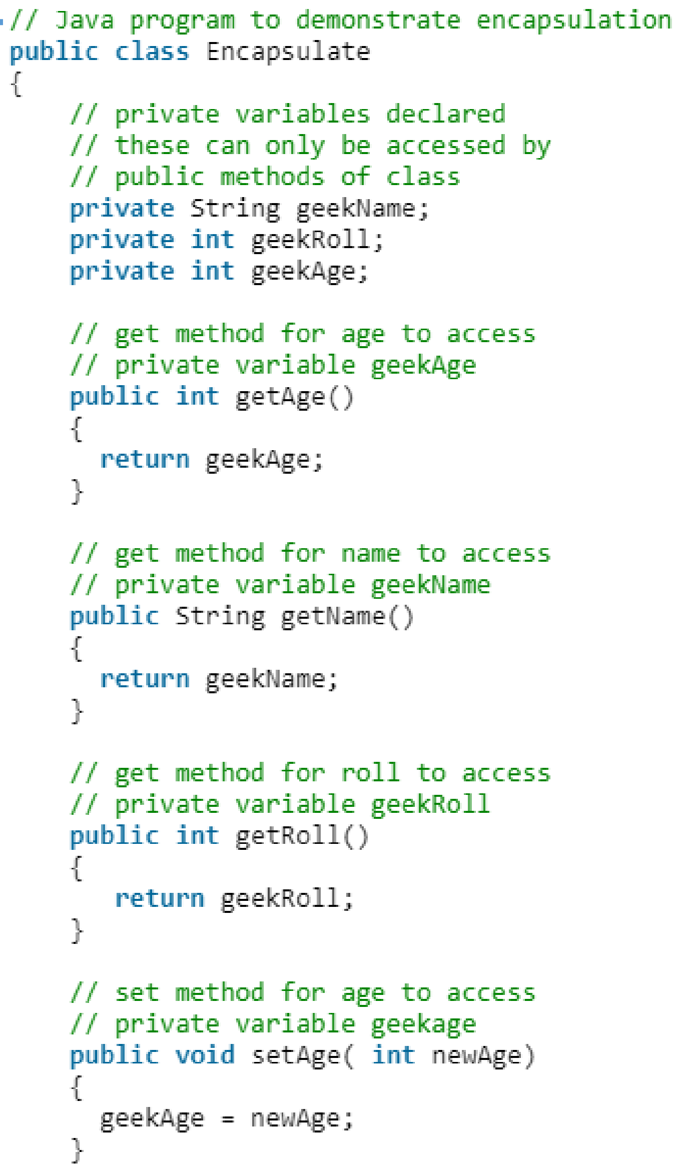
Polymorphism Example

```java
// Java program to demonstrate Polymorphism

// This class will contain
// 3 methods with same name,
// yet the program will
// compile & run successfully
public class Sum {

    // Overloaded sum().
    // This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum().
    // This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum().
    // This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

- **Encapsulation**

  - Describes the idea of wrapping data and the methods that work on data within one unit, e.g., a class in Java. This concept is often used to hide the internal state representation of an object from the outside.

Encapsulation Example



```
// Java program to demonstrate encapsulation
public class Encapsulate
{
    // private variables declared
    // these can only be accessed by
    // public methods of class
    private String geekName;
    private int geekRoll;
    private int geekAge;

    // get method for age to access
    // private variable geekAge
    public int getAge()
    {
      return geekAge;
    }

    // get method for name to access
    // private variable geekName
    public String getName()
    {
      return geekName;
    }

    // get method for roll to access
    // private variable geekRoll
    public int getRoll()
    {
       return geekRoll;
    }

    // set method for age to access
    // private variable geekage
    public void setAge( int newAge)
    {
      geekAge = newAge;
    }
```

- **Abstraction**

- Abstraction is the process of refining away all the unneeded/unimportant attributes of an object and keep only the characteristics are most suitable for your domain

Abstraction Example

```java
// Java program to illustrate the
// concept of Abstraction
abstract class Shape
{
    String color;

    // these are abstract methods
    abstract double area();
    public abstract String toString();

    // abstract class can have constructor
    public Shape(String color) {
        System.out.println("Shape constructor called");
        this.color = color;
    }

    // this is a concrete method
    public String getColor() {
        return color;
    }
}
class Circle extends Shape
{
    double radius;

    public Circle(String color,double radius) {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * Math.pow(radius, 2);
    }

    @Override
    public String toString() {
        return "Circle color is " + super.color +
                       "and area is : " + area();
    }

}
```

```java
class Rectangle extends Shape{

    double length;
    double width;

    public Rectangle(String color,double length,double width) {
        // calling Shape constructor
        super(color);
        System.out.println("Rectangle constructor called");
        this.length = length;
        this.width = width;
    }

    @Override
    double area() {
        return length*width;
    }

    @Override
    public String toString() {
        return "Rectangle color is " + super.color +
                           "and area is : " + area();
    }

}
public class Test
{
    public static void main(String[] args)
    {
        Shape s1 = new Circle("Red", 2.2);
        Shape s2 = new Rectangle("Yellow", 2, 4);

        System.out.println(s1.toString());
        System.out.println(s2.toString());
    }
}
```

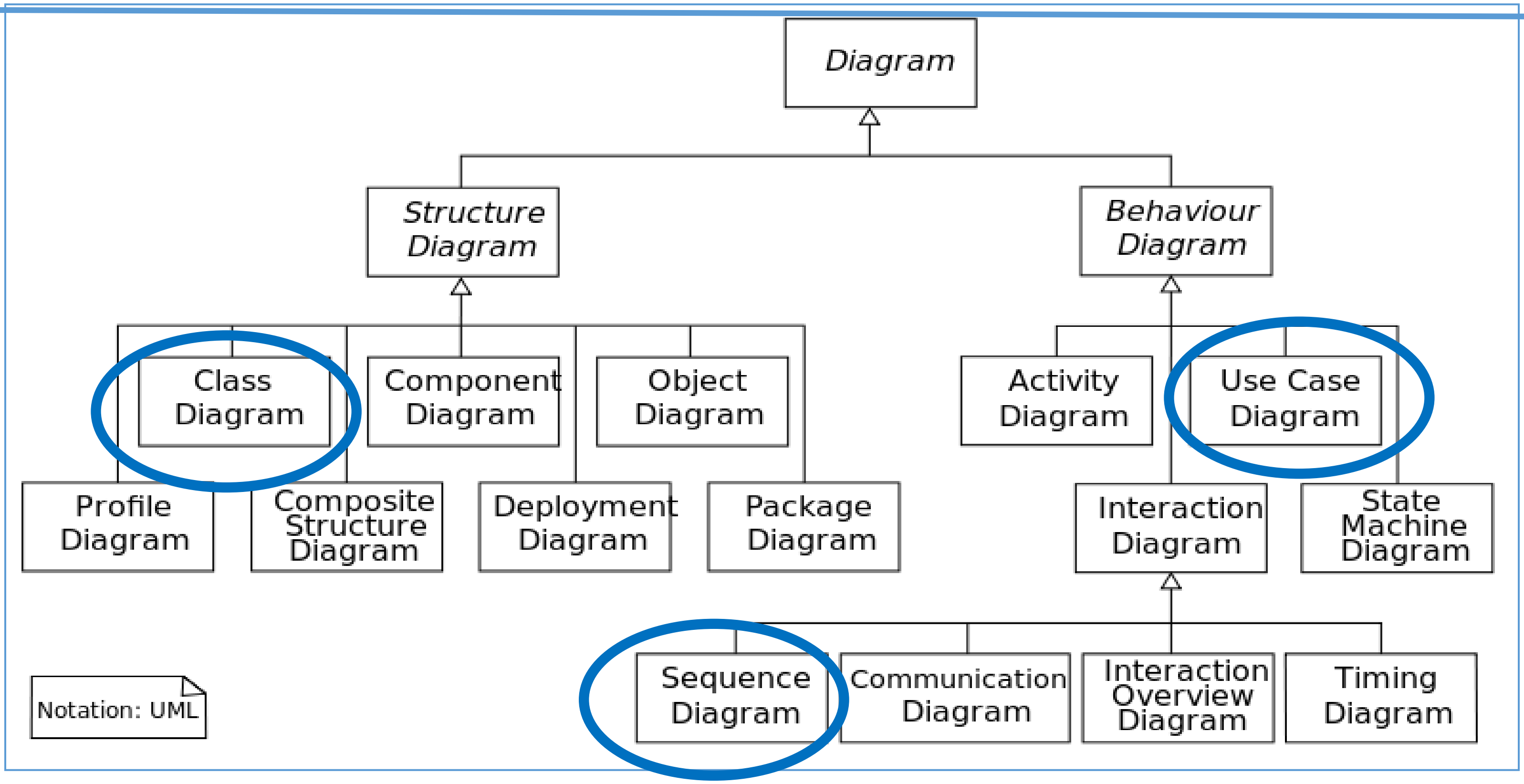
UML Diagrams

What is UML ?

- UML (Unified Modeling Language)
    - An emerging standard for modeling object-oriented software.
    - includes graphic notation techniques to create visual models of
    - object-oriented software-intensive systems

- Reference: “The Unified Modeling Language User Guide”, Addison Wesley, 1999.

UML diagrams

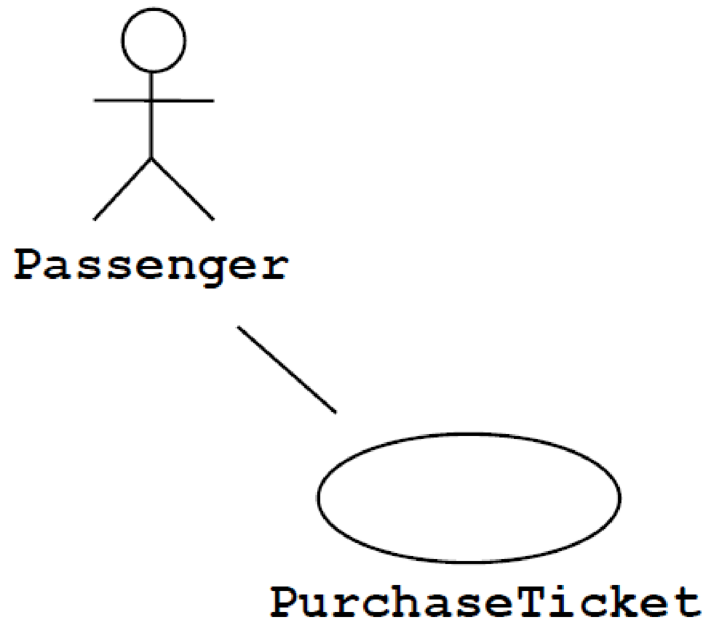


Use Case diagram: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use case descriptions.

Class diagram (Static structure diagram**)**: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.

Sequence diagram and Communication diagram: shows how objects communicate with each other in terms of a sequence of messages.

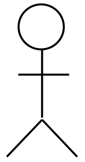
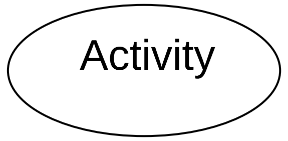
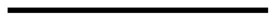
Use Case Diagrams



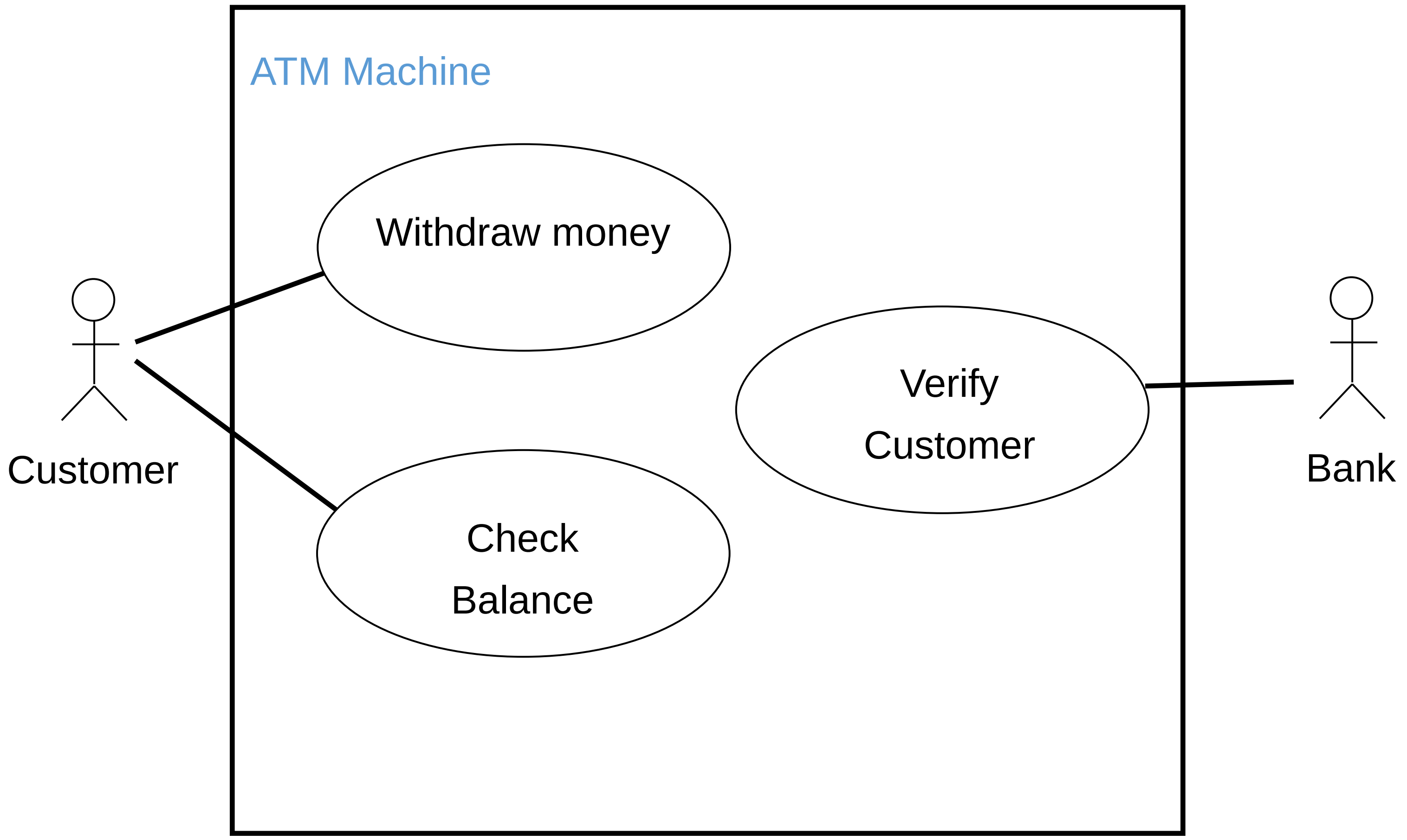
- Used during requirements elicitation to represent external behavior
- ***Actors*** represent roles, that is, a type of user of the system
- ***Use cases*** represent a sequence of interaction for a type of functionality

- The use case model is the set of all use cases. It is a complete description of the functionality of the system and its environment

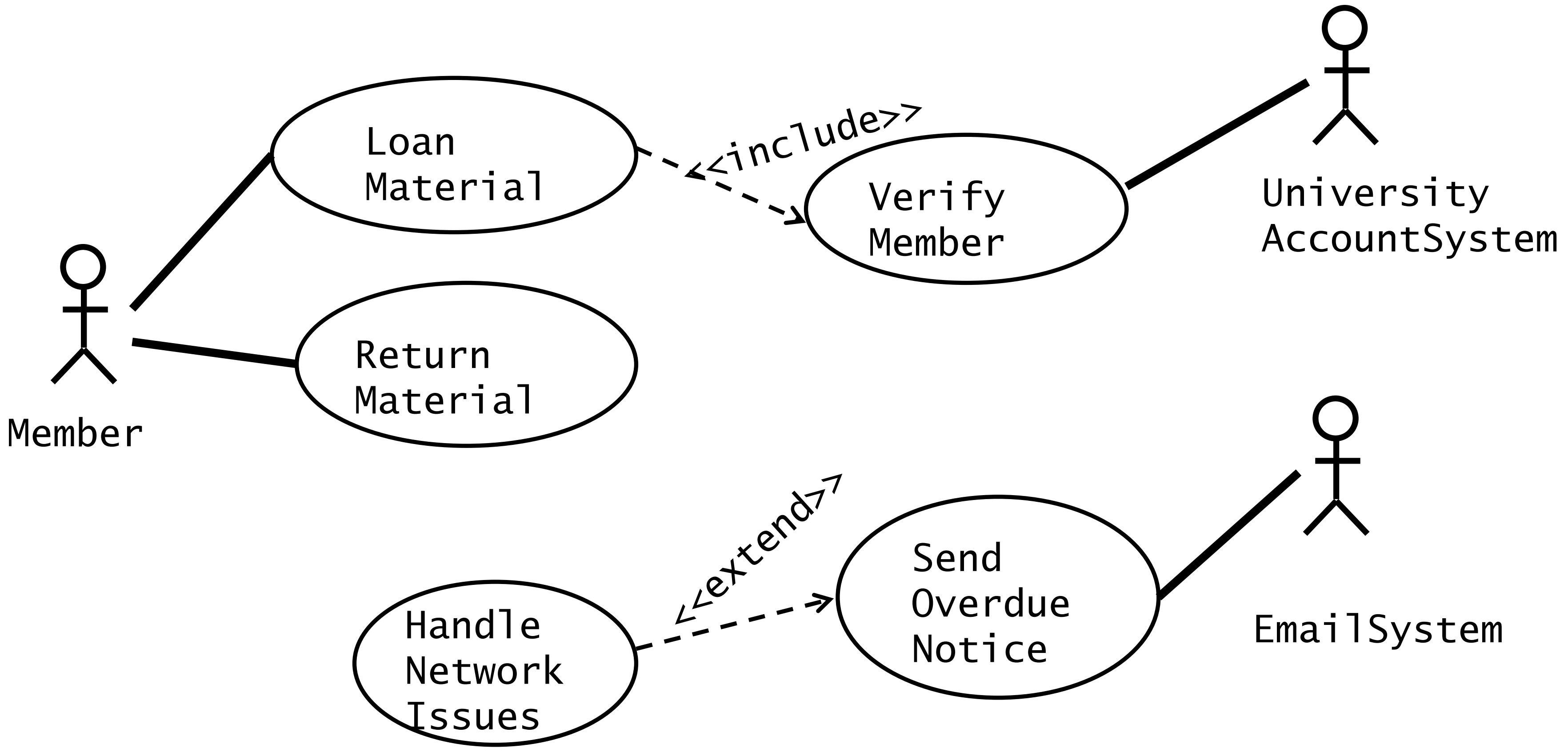
Use Case Diagram Elements

Name	Notation	Description
Actor		An external entity that interacts with system
Use Case		Unit of functionality performed by system, which yields result / value for Actor
Association		Connects Actor to Use Cases(s) in which they participate

Example of Use Case Diagram



Example of Use Case Diagram



Class Diagrams

- Class diagrams represent the structure of the system

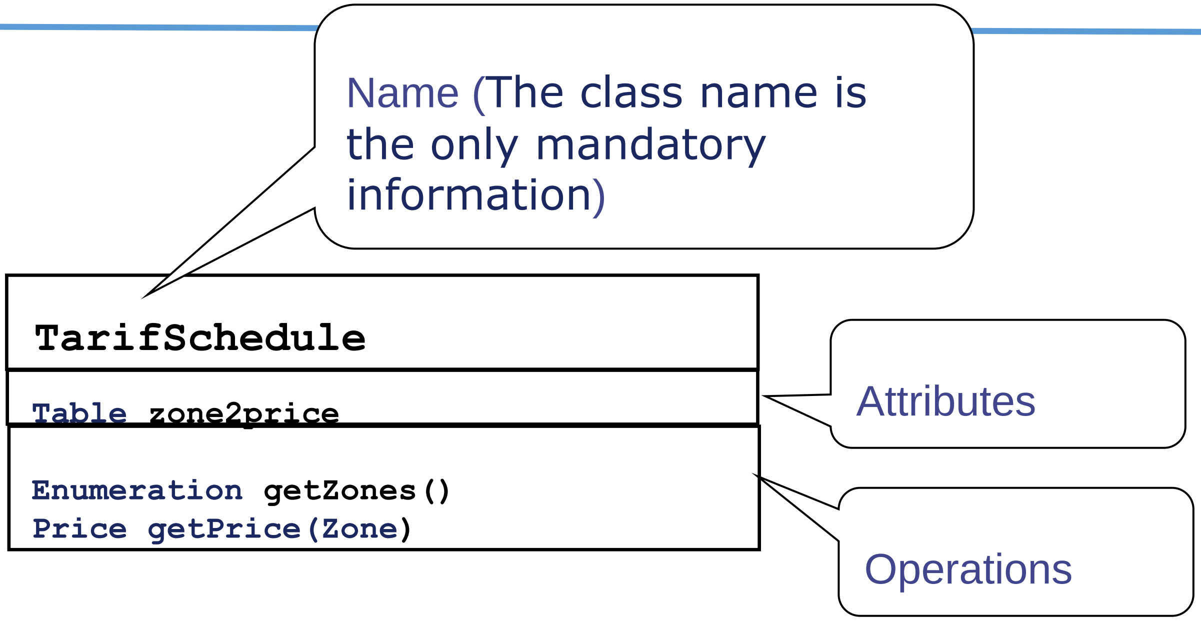
- The classes define the responsibilities for doing various activities

- Used during
  - requirements analysis: model application domain concepts
  - system design: model subsystems
  - object design: specify the detailed behavior and attributes of classes

Essentials of Class Diagrams

- The main symbols shown on class diagrams are:
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

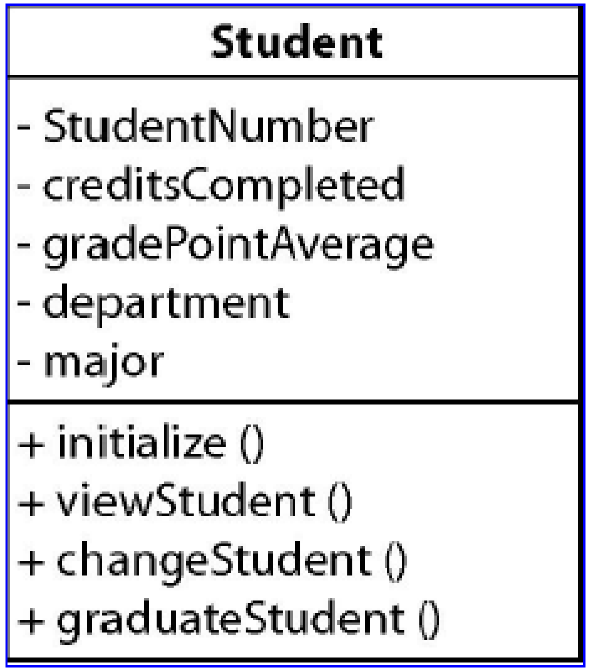
class



- A ***class*** represents a concept
- A class encapsulates states ***(attributes)*** and behaviour ***(operations)***

Each attribute has a *type*
Each operation has a *signature*

Class Visibility

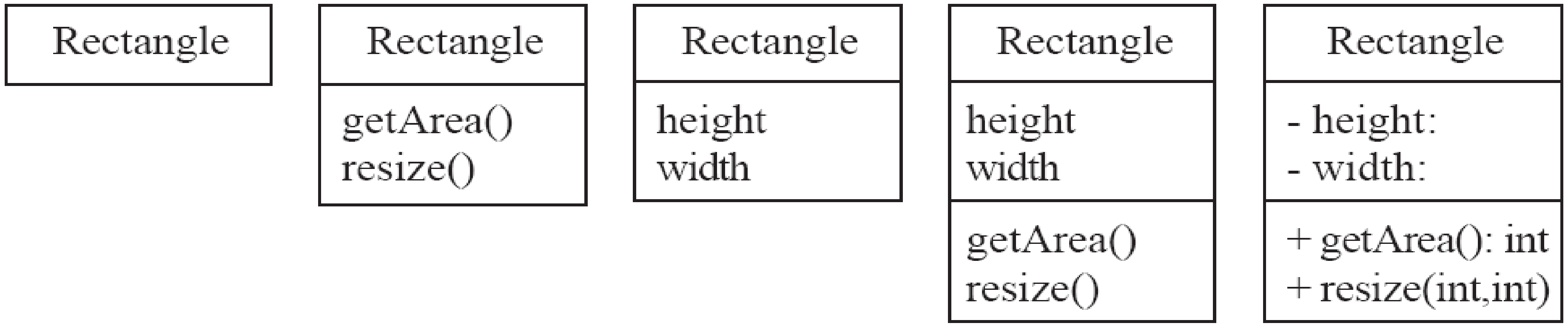


+: Public
-: Private
#: Protected

Classes

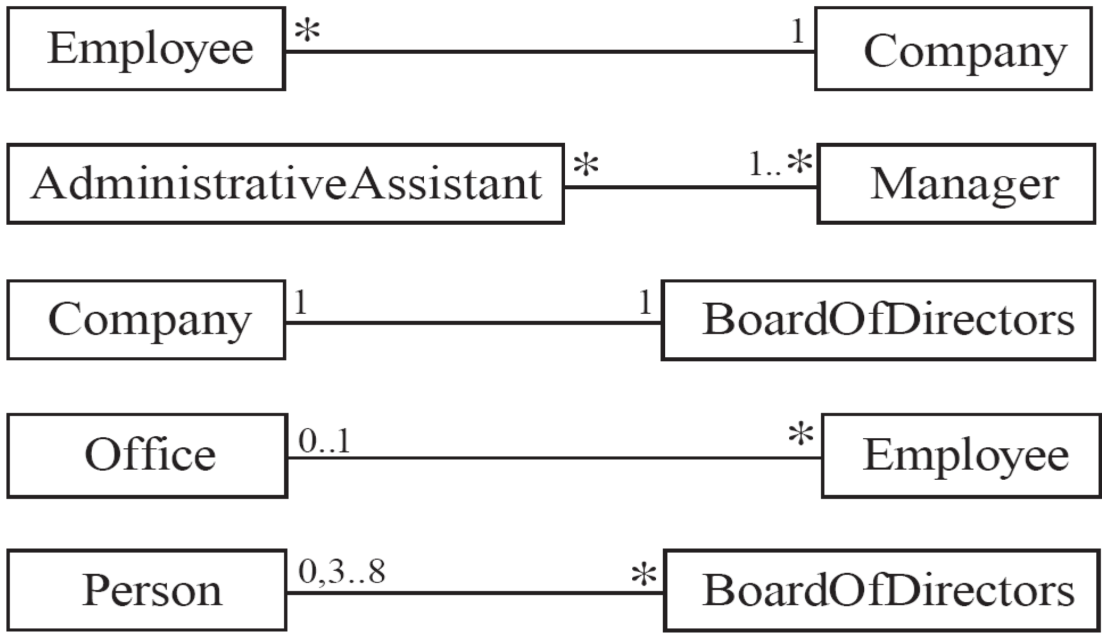
- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType



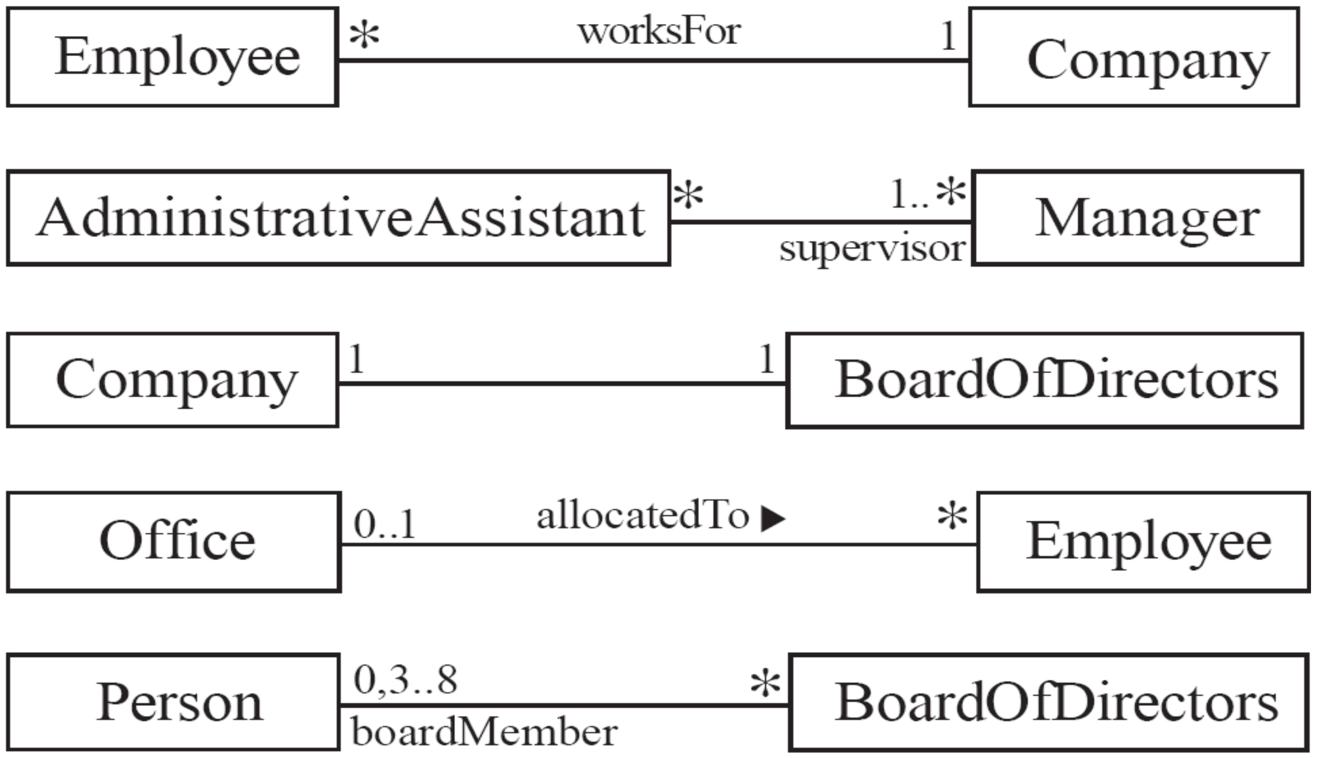
Associations and Multiplicity

- An association is used to show how two classes are related to each other
  - Symbols indicating *multiplicity* are shown at each end of the association



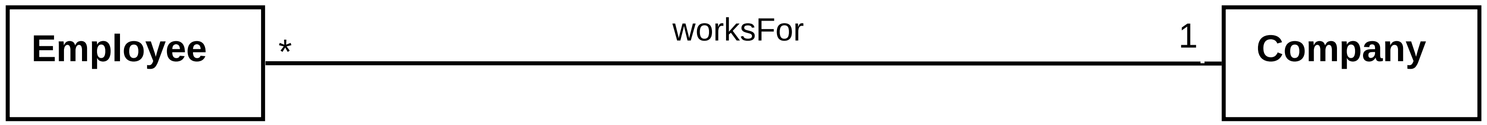
Labelling associations

- Each association can be labelled, to make explicit the nature of the association



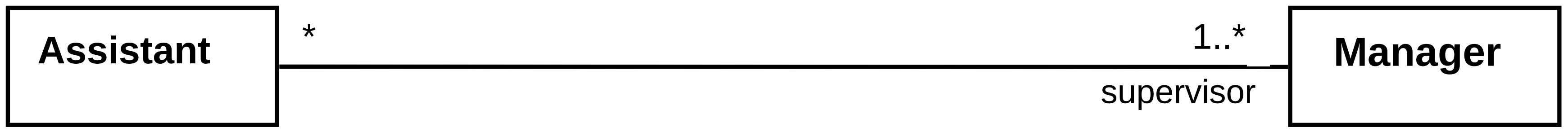
Analyzing and validating associations

- **Many-to-one**
  - A company has many employees,
  - An employee can only work for one company.
  - A company can have zero employees
  - It is not possible to be an employee unless you work for a company



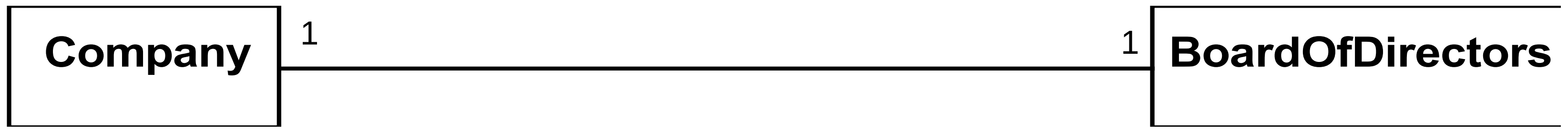
Analyzing and validating associations

- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Some managers might have zero assistants.
  - An assistant should have at least one manager



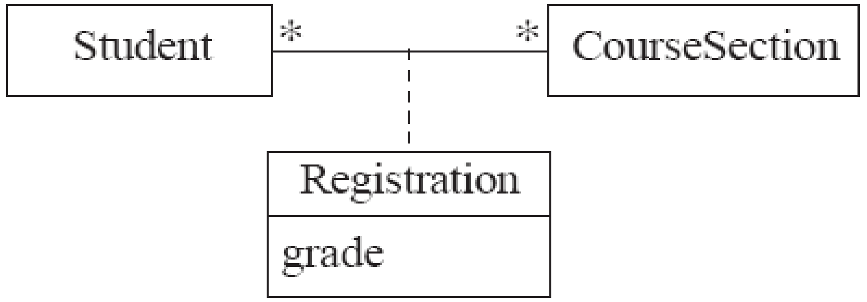
Analyzing and validating associations

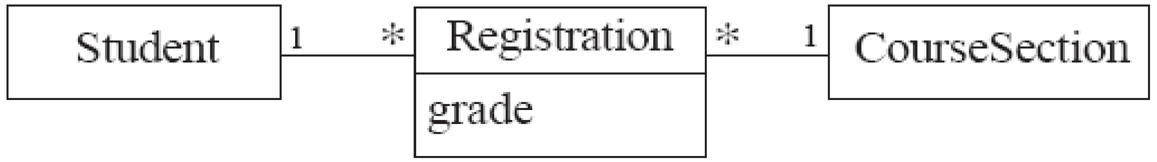
- **One-to-one**
  - For each company, there is exactly one board of directors
  - A board is the board of only one company
  - A company must always have a board
  - A board must always be of some company



Association classes

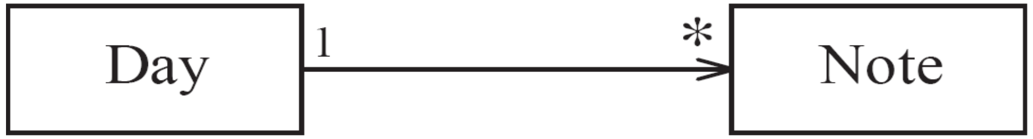
- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent





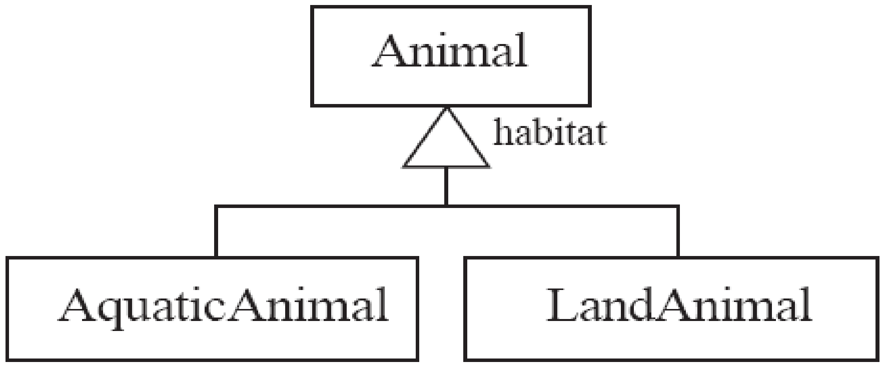
Directionality in associations

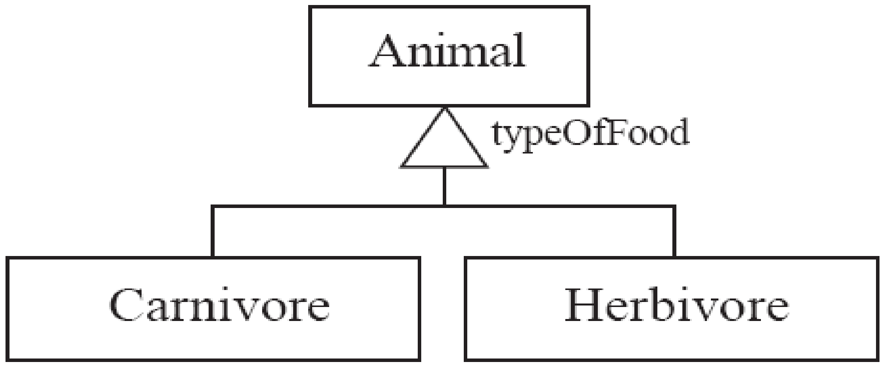
- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end



Generalization

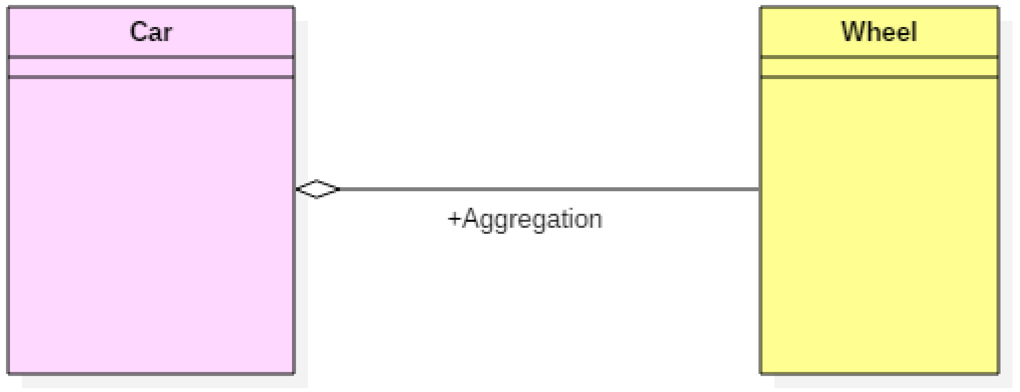
- Specializing a superclass into two or more subclasses
  - A *generalization set* is a labelled group of generalizations with a common superclass
  - The label (sometimes called the *discriminator*) describes the criteria used in the specialization





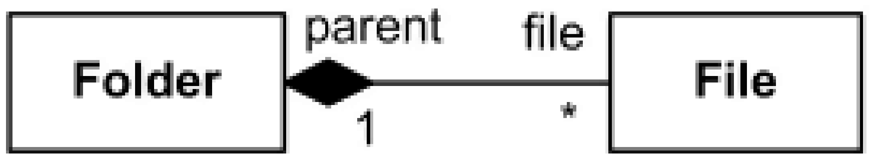
More Advanced Features: Aggregation

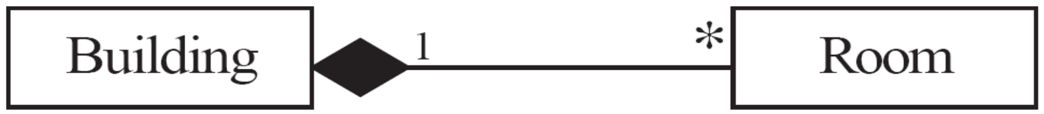
- An aggregation is a subtype of an association relationship.
  - Can be described in simple words as "an object of one class can own or access the objects of another class."



Composition

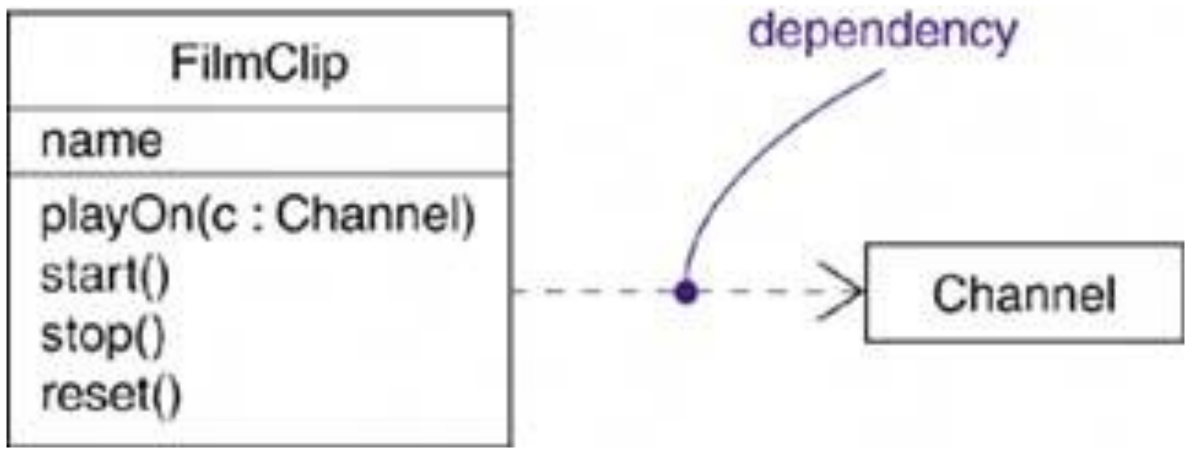
- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well



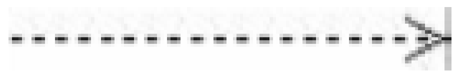
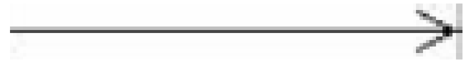
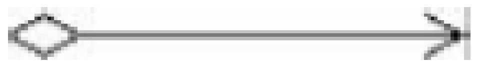
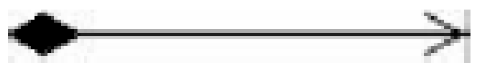
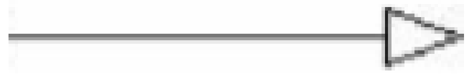


Dependency

- A dependency is a relationship that states that one uses the information and services of another thing, but not necessarily the reverse.



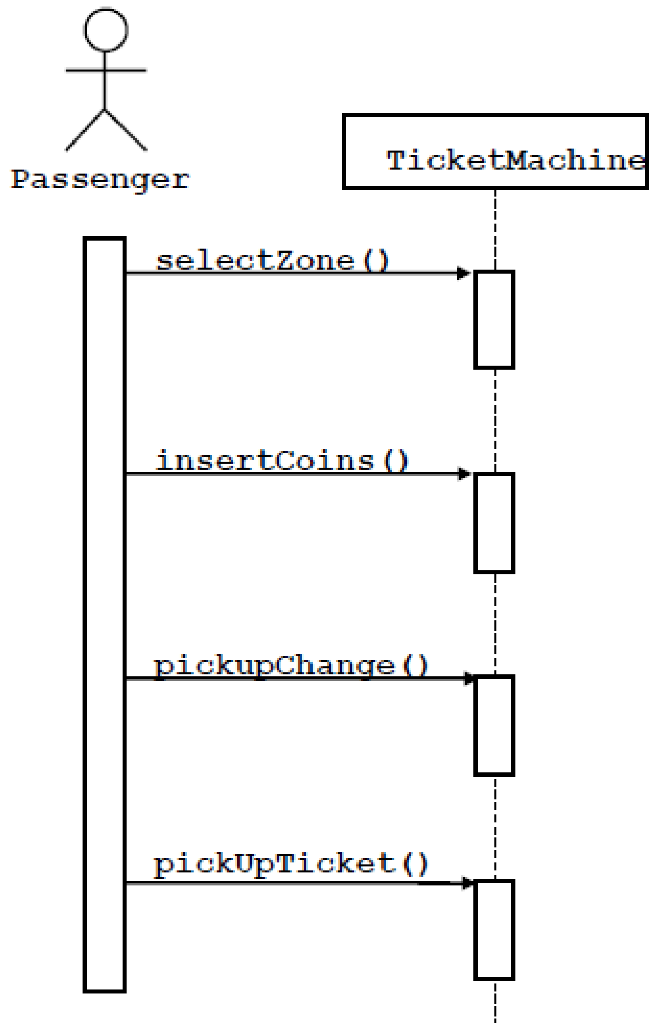
Class Diagram Relationship Summary

Relationship	<any line>	meaning
Dependency		"uses a"
Association		"has a"
Aggregation		"owns a"
Composition		"is composed of"
Generalization		"is a"

Examples

- Modem and Keyboard are devices
- A file is either an ordinary file or a directory file. Directories contain files
- A polygon consists of at least three points
- Universities employ persons for a period
- An email can be sent to one or more recipients with a title, text and attached files

UML Sequence diagrams



- Focuses on the time-ordering of messages between objects
- Used during requirements analysis
  - To refine use case descriptions
  - to find additional objects ("participating objects")
- Used during system design
  - to refine subsystem interfaces
- **Classes** are represented by rectangles
- **Activations** are represented by narrow rectangles
- **Lifelines** are represented by dashed lines
- **Messages** are represented by arrows

Metrics in uml

Metrics in UML

Number of atomic Scenarios

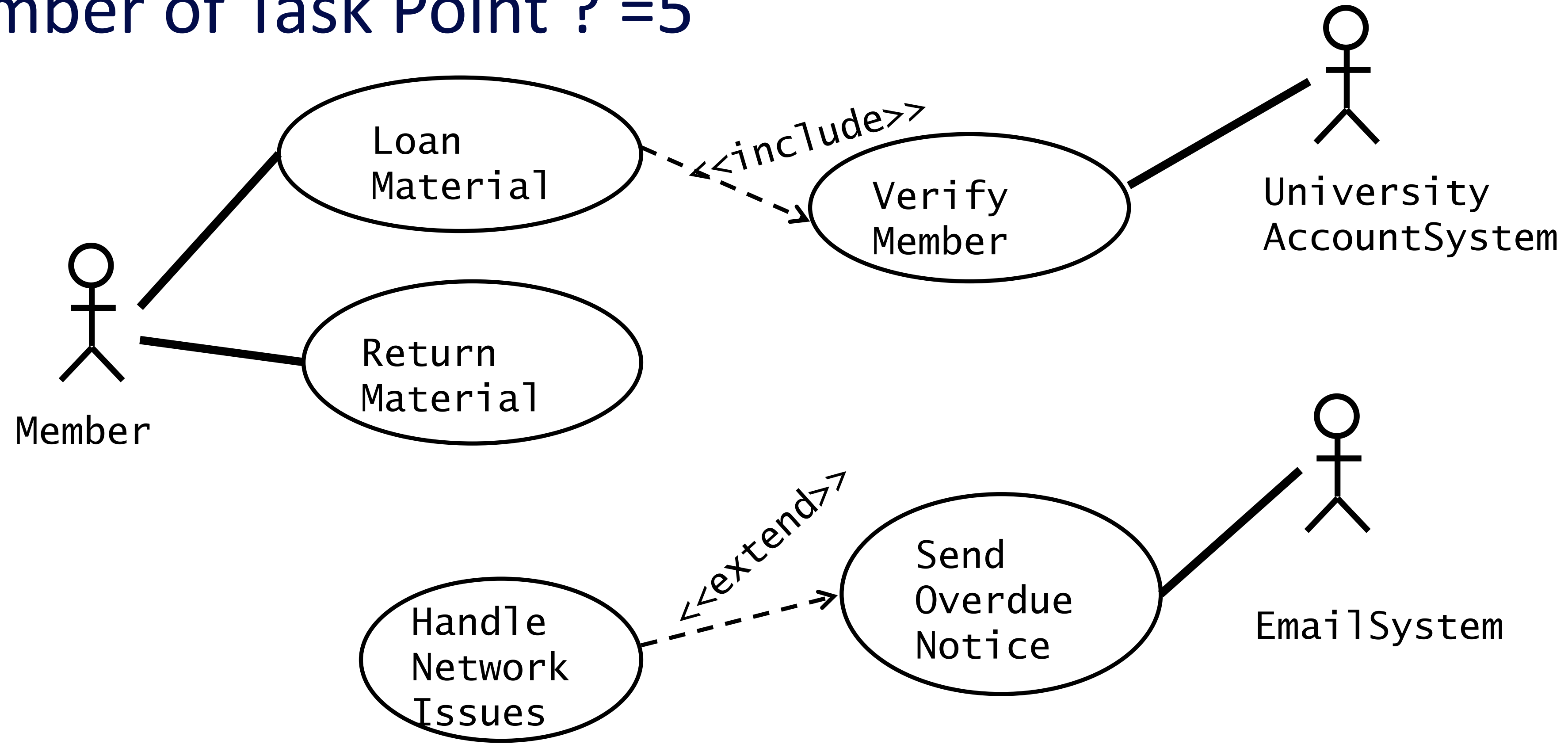
- **Name and origin**: Task Points (#TP) (Graham, 1995).
- **Measurement:** Task Points are the simple count of the so-called "atomic" task scripts (more formal form of use cases) in an object-oriented analysis model.

- **Usage:** Task Points are suggested for tracking and estimating the overall software proves.

it does not include any structural information about the system.

Example: #TP

- Number of Task Point ? =5



Number of Classes

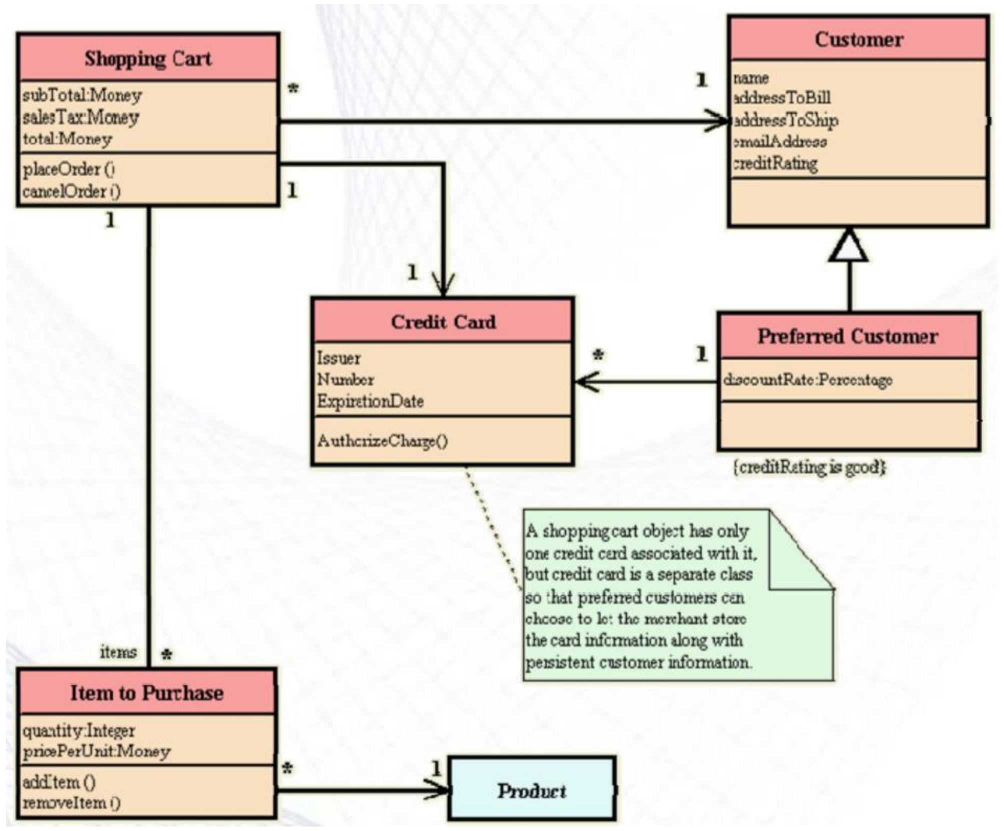
- **Name and origin**: #c (Lorenz and Kidd, 1994).
- **Measurement**: #c is the number of classes that make up a system.

- **Usage**: The number of design classes can be used to track the process of design

Similar to #TP it ignores the association and aggregation among classes. Does not bear any structural meaning

Example : #c

- Number of Classes ? = ?



Number of Instance Methods (#im)

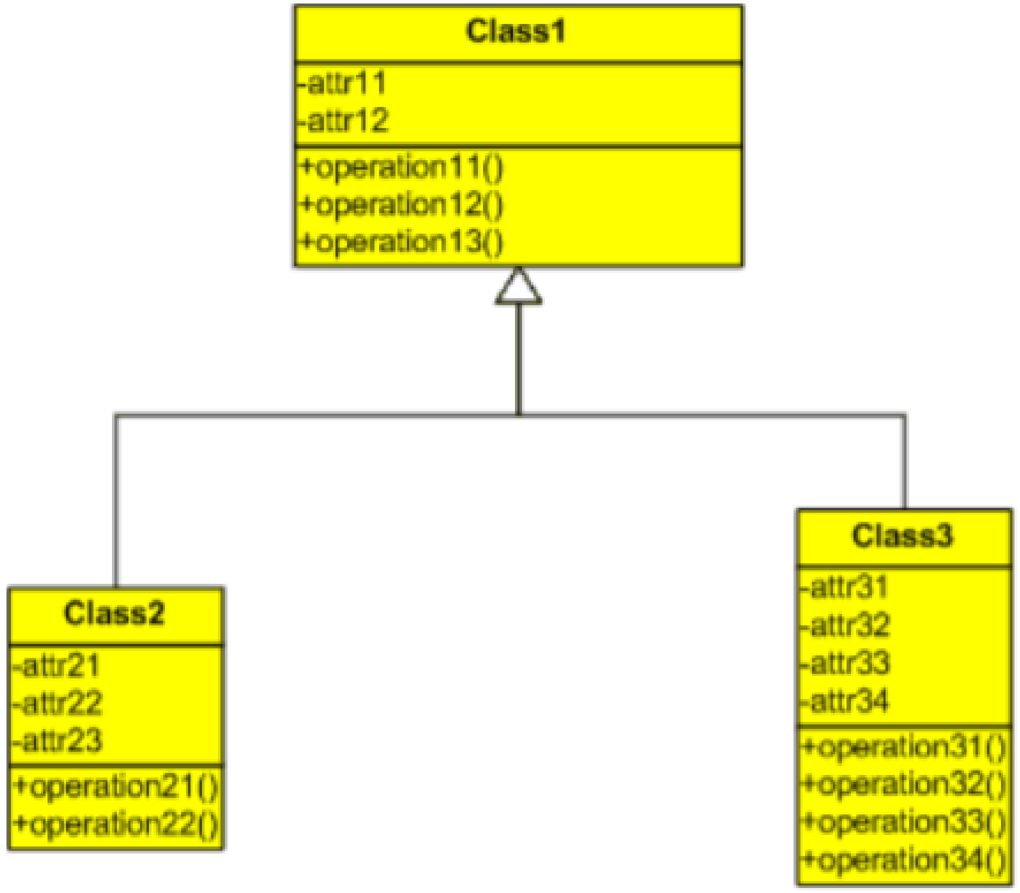
- **Name and origin**: #im (Lorenz and Kidd, 1994)
- **Measurement**: #im is the number of methods that siblings of a class can understand (regardless of eventual method access restrictions such as “private” or “protected”).
- **Usage**: A system-wide summation of all #im values can be used as measure of design size

Number of Class Methods (#cm)

- **Name and origin**: #cm (Lorenz and Kidd, 1994).
- **Measurement**: #cm the number of methods that a class can understand (regardless of eventual method access restriction such as “private” or “protected”).

- **Usage**: : A system-wide summation of all #im values can be used as measure of design size

- For Class 1
  - #cm = ?
- For Class 3
  - #im= 7

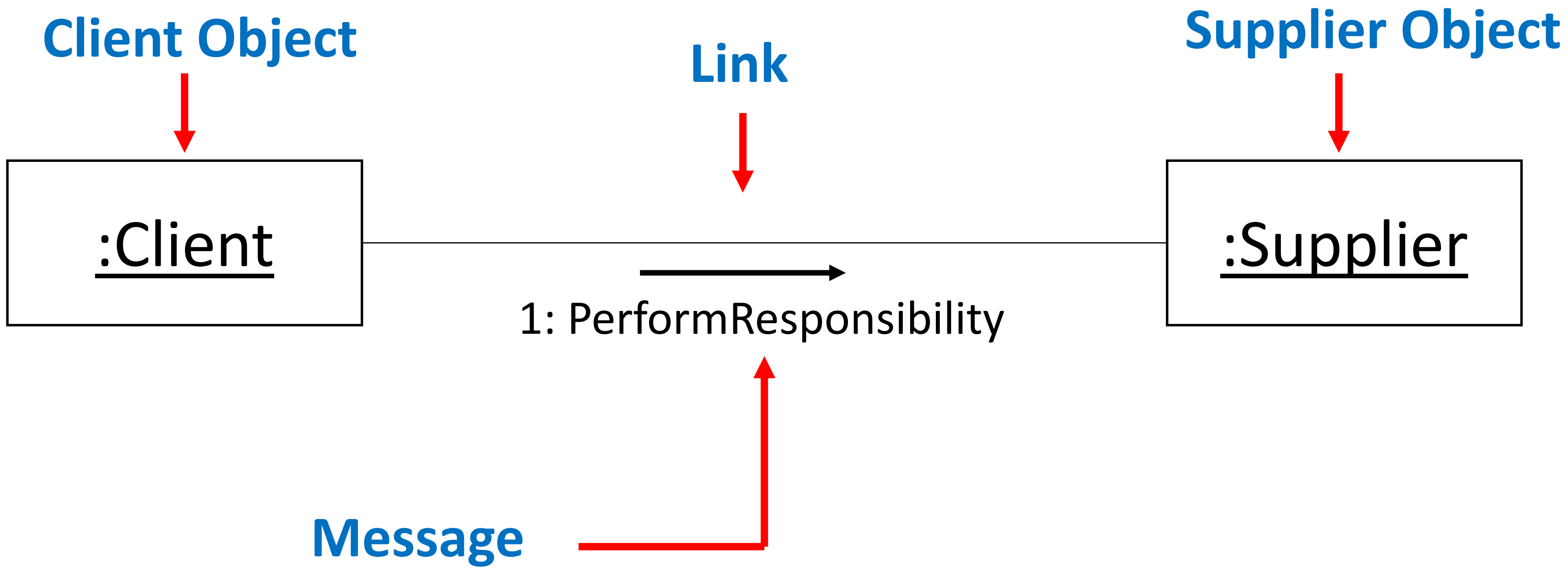


Coupling

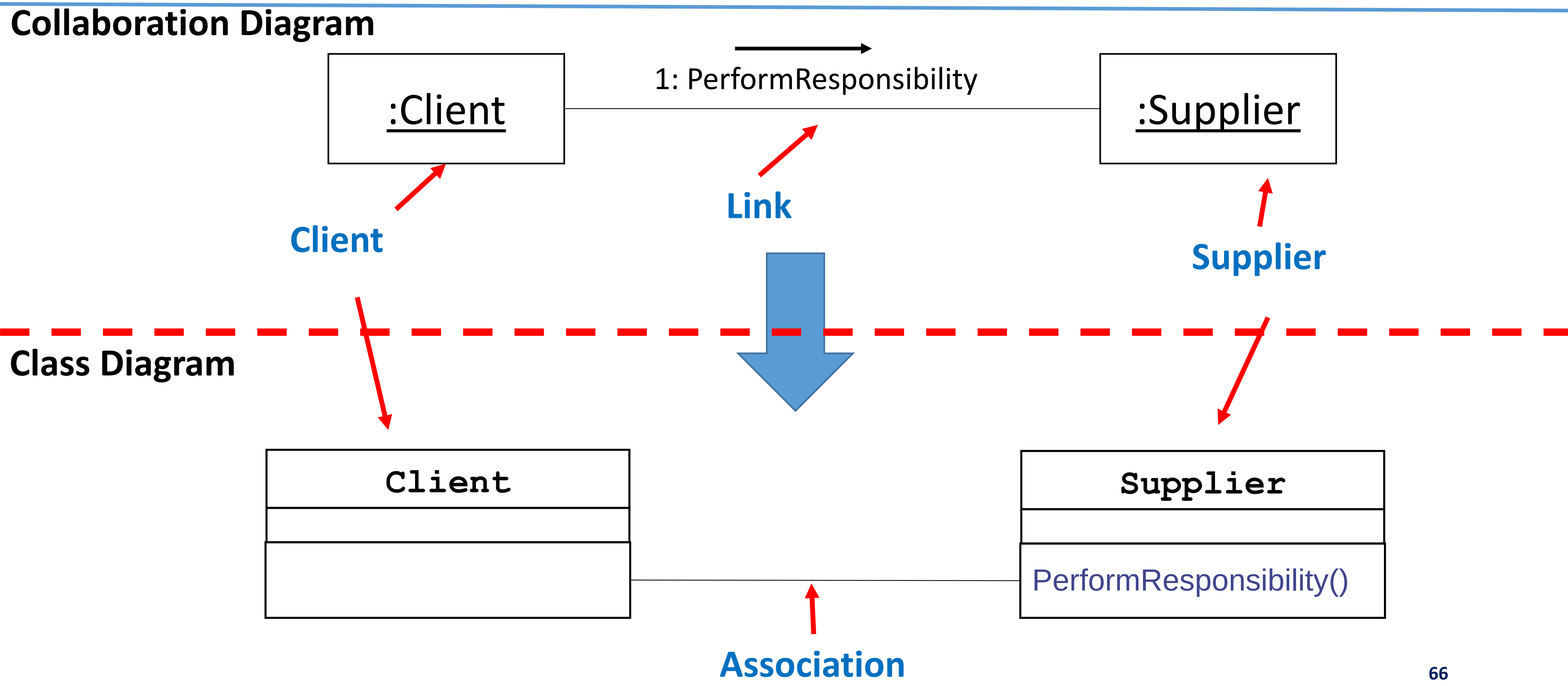
- Coupling describes how strongly one element (class / package) relates to another element
- **Coupling between objects** is a count of the number of “links” between them.
- **Coupling between classes** is a count of the number of other classes to which a class is related. It is measured by counting the number of distinct “associations” with the other classes.
- **Coupling between Packages** is a count of the number of distinct “associations” between packages.

What are Links and Associations

- An object is an instance of a class
- In the same way, a “link” is an instance of an “association”

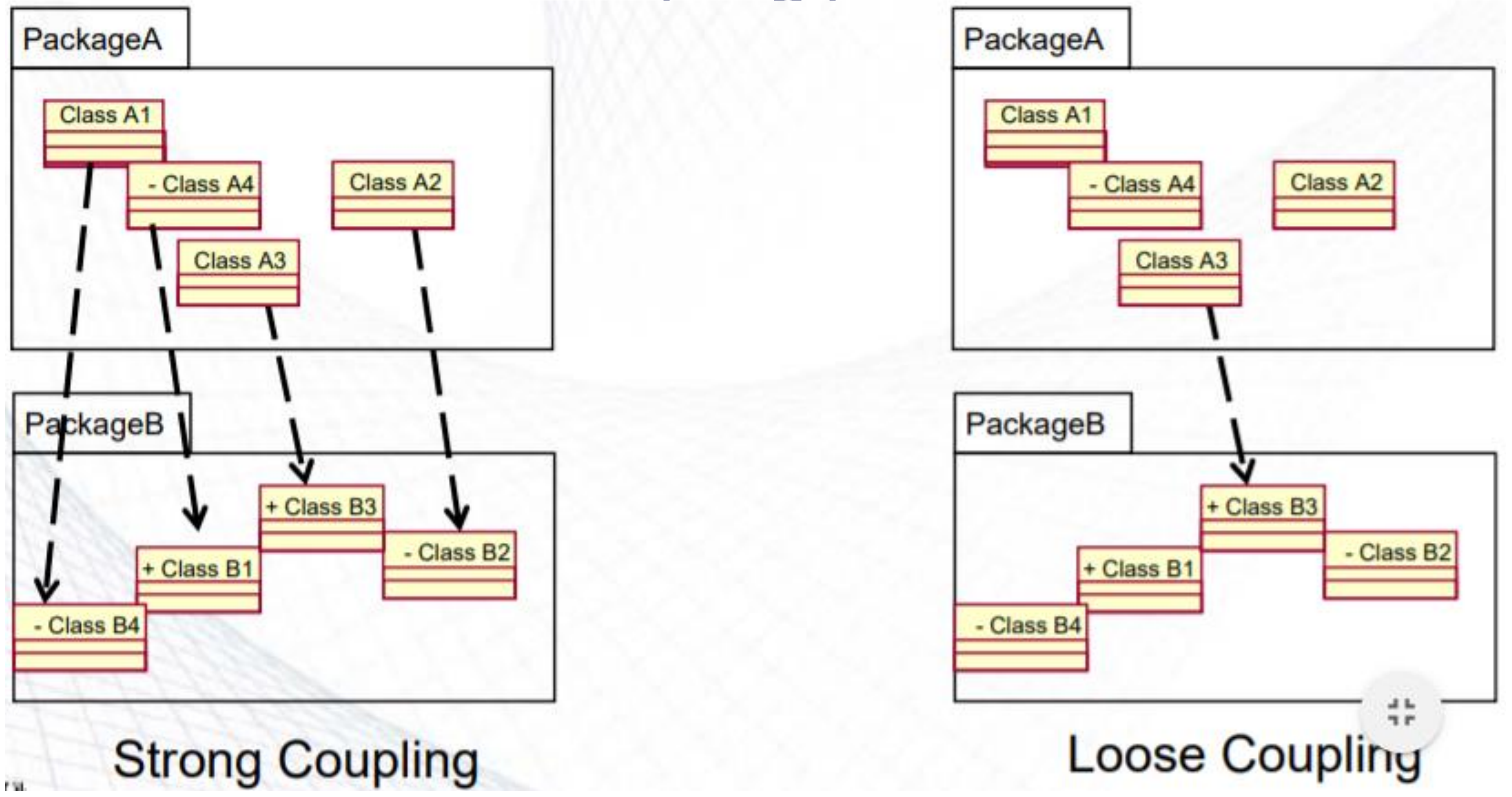


What are links and associations ?



Package Coupling: Class Relationships

- Strive for the loosest coupling possible



Cohesion

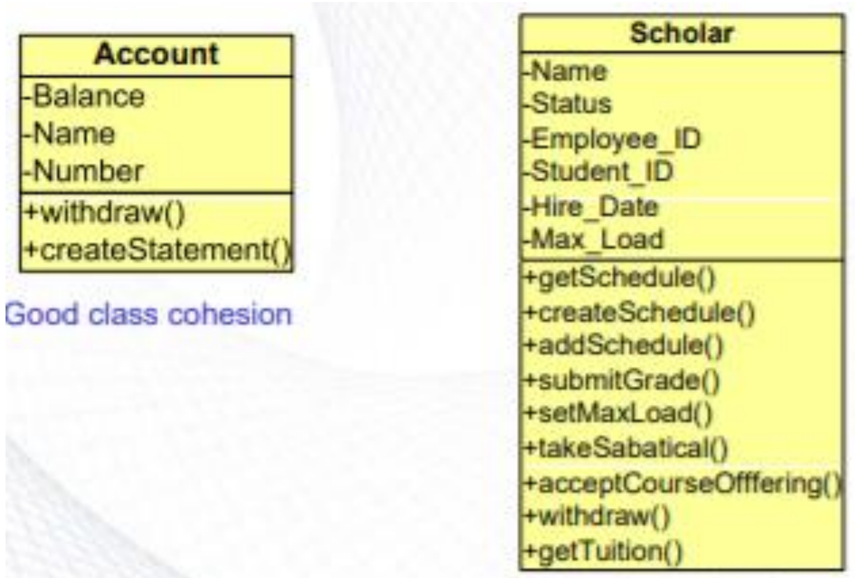
- Cohesion describes how strongly related the responsibilities between design elements can be described

- The goal is to achieve “high cohesion”
    - High cohesion between classes is when responsibilities are highly related

Cohesion: How to Measure?

- Find “similarities” and “dissimilarities” between operations (methods) in a class.
- Remember a “link” will eventually be translated into an “operation”
- Used diagram to measure cohesion : Class diagrams

Example: Cohesion

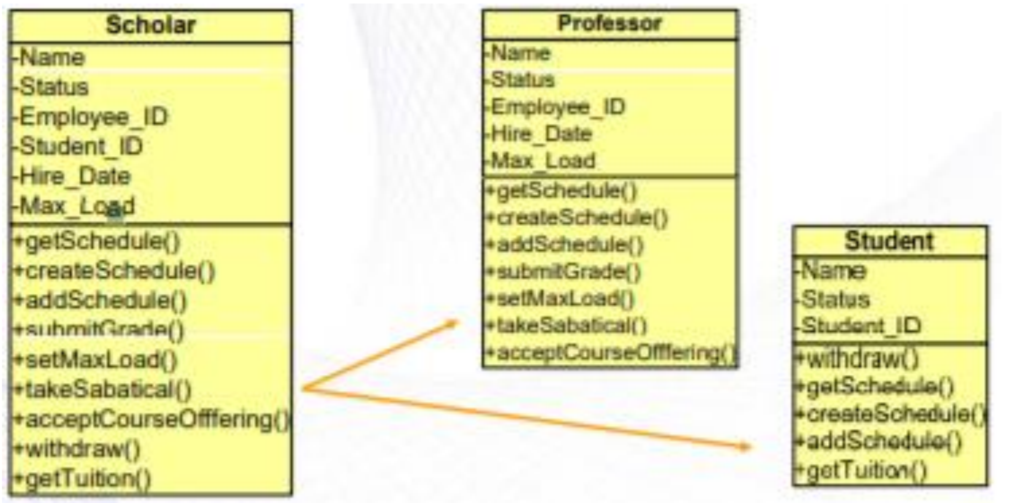


Good class cohesion

Bad class cohesion

Cohesion can help split and merge

The classes



OO Project metrics

- What we want to measure in an OO project?

  - Number of Classes, Operations (Methods), Attributes (Variables)
    - Lines Of Code (LOC) and Statement Count Total and/or Averaged by class and/or method

  - Structural measurement:
    - Coupling, Cohesion

OO package metrics

- What we want to measure for a package?
  - Number of classes, operations (methods), attributes (variables), Average by class and/or method
- Structural measurement
  - Coupling, Cohesion
  - Maximum Inheritance Depth

OO Class Metrics

- What we want to measure for a class
  - Number Attributes and Operations
  - Lines of code (LOC) and statement count
  - Inheritance related metrics
  - Collaborators (Cohesion and Coupling related metrics)

OO Attribute Metrics

- What we want to measure for an attribute?
  - How many times used

OO Operation Metrics

- What we want to measure for an operation?
  - Number of local variables
  - Lines of code (LOC) and statement count
  - Cyclomatic Complexity

Lines of code

- The most commonly used measure of source code program length is the number of lines of code (LOC).
  - NCLOC : non-commented source line of code or effective lines of code/
  - CLOC : commented source line of code.
- By measuring NCLOC and CLOC separately we can define:
  - Total length (LOC) = NCLOC + CLOC
- The ratio: CLOC/LOC measures the density of comments in a program
- Variations of LOC:
  - Count of physical lines including blank lines
  - Count of all lines except blank lines and comments.
  - Count of all statements except comments (statements taking more than one line count as only one line
  - Count of only executable statements, not including exception conditions

Lines of code

- Easy to measure; but *not well-defined* for modern languages
  - What's a line of code? (vague definition)
  - Language dependability
  - Not available for early planning
  - Developers' skill dependability
- A poor indicator of productivity
  - Ignores software reuse, code duplication, benefits of redesign

Chidamber and Kemerer OO Metrics (CK - 1994)

- Weighted methods per class (MWC)
- Depth of inheritance tree (DIT)
- Number of children (NOC)
- Coupling between object classes (CBO)
- Response for class (RFC)
- Lack of cohesion metric (LCOM)

Weighted methods per class (WMC)

- c_i is the *complexity* of each method M_i of the class
  - Often, only public methods are considered

- If methods complexity is similar, then, by applying unity, this metric would simply count the number of methods per class

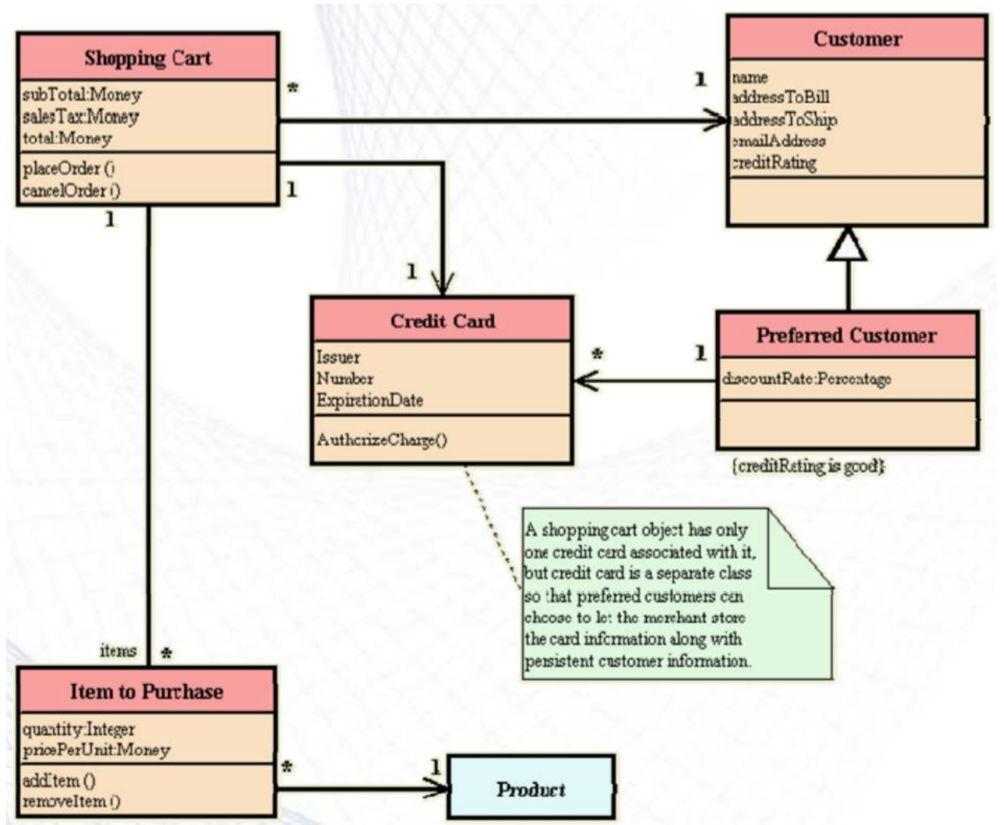
$$WMC = \sum_{i=1}^{n} c_i$$

- The number of methods and complexity of methods involved is a direct predictor of how much time and effort is required to develop and maintain the class.

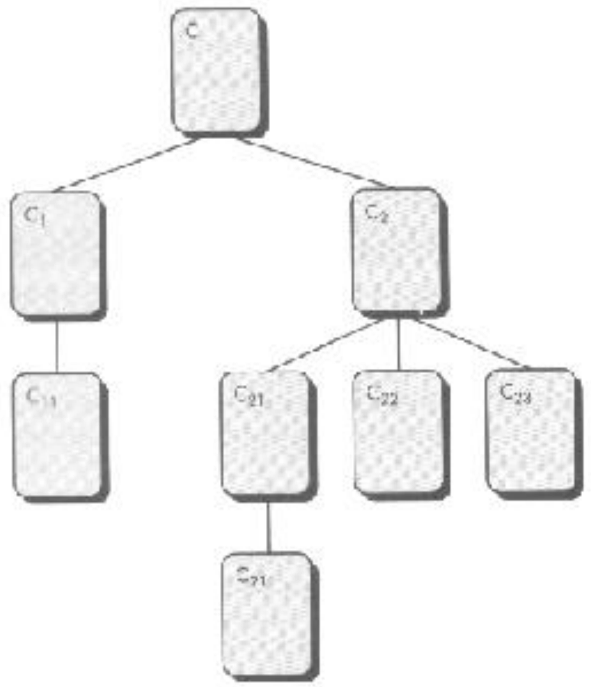
Smaller value are better

Weighted methods per class (WMC)

- Example:
  - WMC for Shopping_Cart = ? WMC for Credit Card = ?



Depth of inheritance tree (DIT)



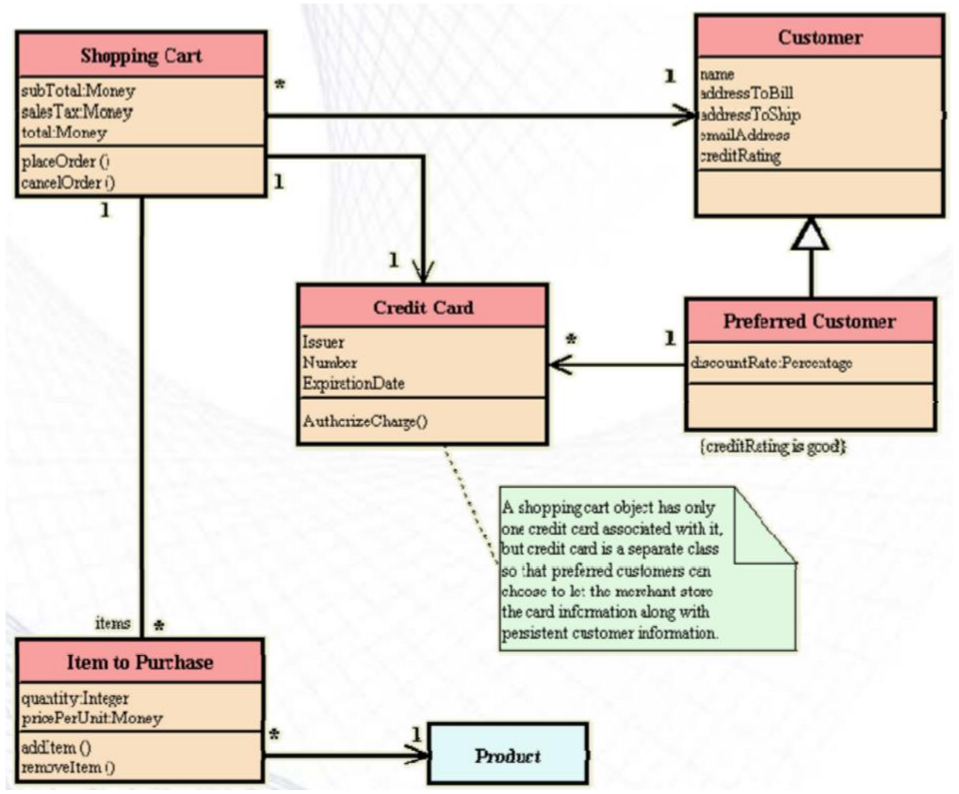
- For the system under examination, consider the hierarchy of classes
- DIT is the *length of the maximum path from the node to the root of the tree*
- Relates to the scope of the properties
  - How many ancestor classes can potential affect a class
- In the preceding example, the number to be entered into the cell would be 4
- The deeper a class is in the hierarchy, the higher the degree of methods inheritance, making it more complex to predict its behavior

Deeper trees constitute greater design complexity, since more methods and classes are involved

Smaller values are better

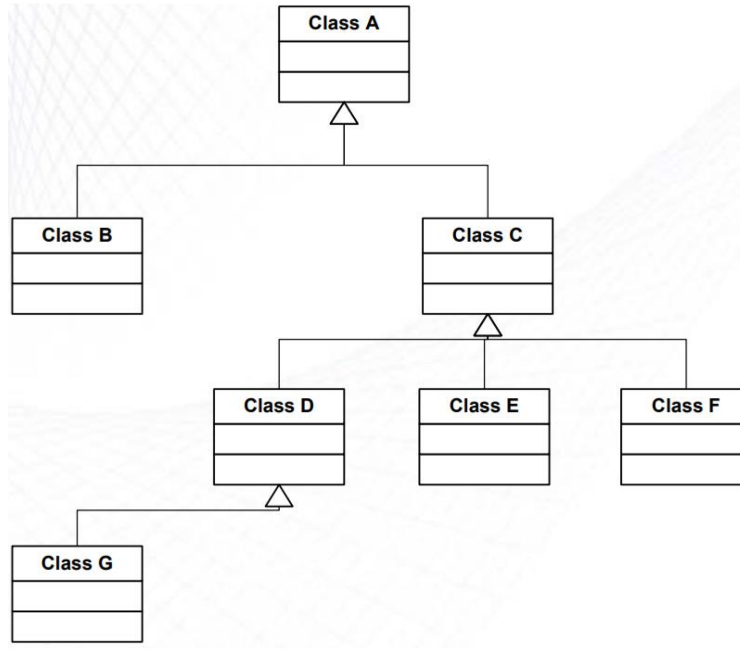
DIT Example 1

- The DIT for *Customer* is ? =
- The DIT for *Preferred_Customer* is ? =

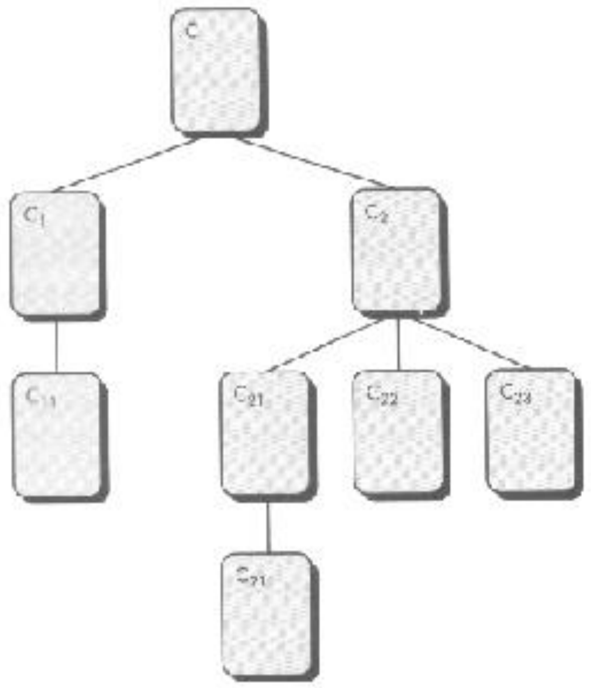


DIT Example 2

- Another example
  - DIT(A)=
  - DIT(B,C)=
  - DIT(D,E,F)=
  - DIT(G)=



Number of children (NOC)

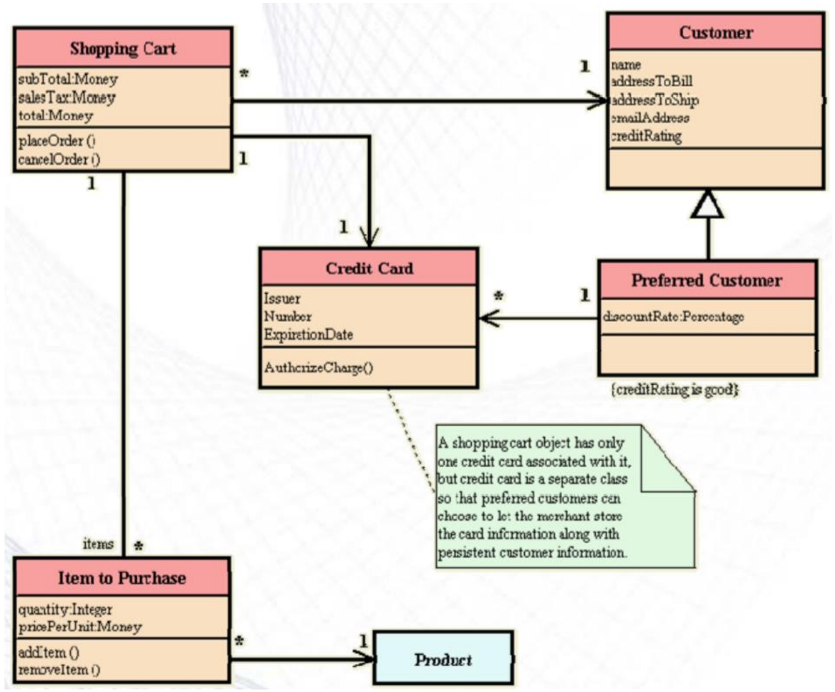


- For any class in the inheritance tree, NOC is the number of *immediate* children of the class
  - The number of direct subclasses
- The greater the number of children, the greater the likelihood of improper abstraction of the parent class
- The number of children gives an idea of the potential influence a class has on the design

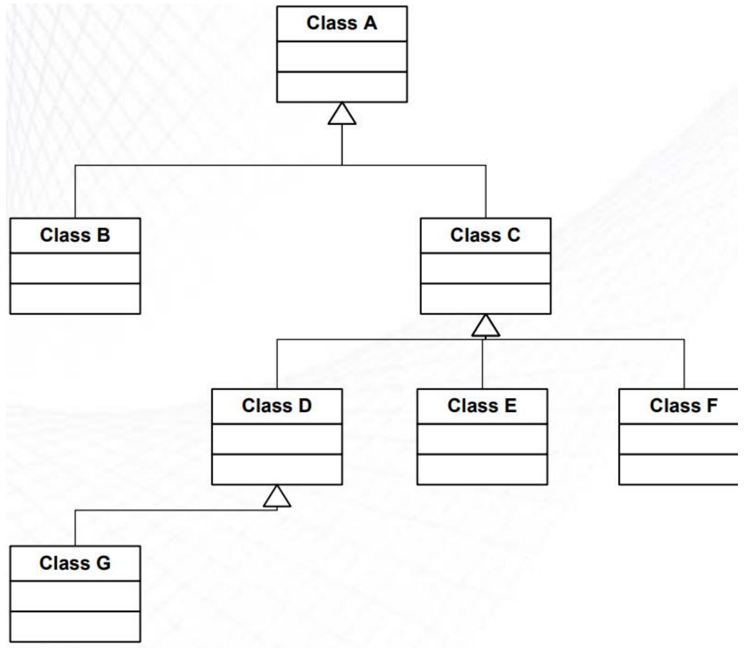
A moderate value indicates scope for reuse and high values may indicate an inappropriate abstraction in the design

Moderate values are better

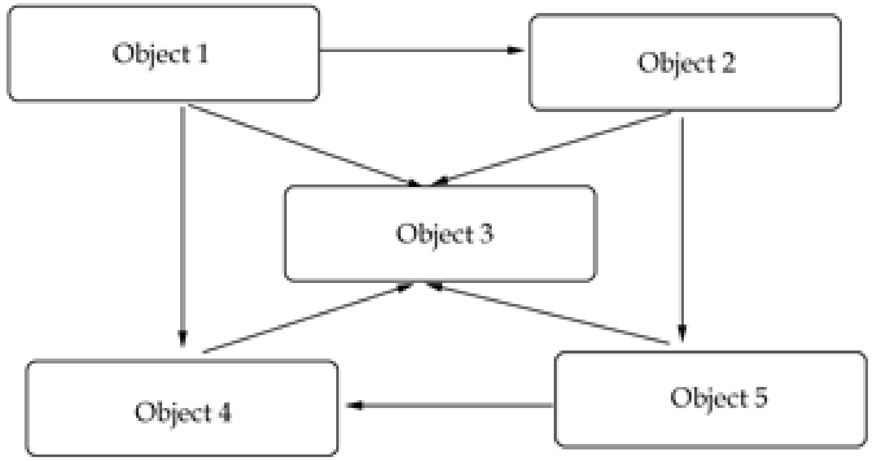
- NOC for Preferred_Customer is ? = 0
- Customer has an NOC of ? = 1



- Another example
  - NoC(A)=?
  - Noc(B)=?
  - NoC(C)=?
  - NoC(D)=?
  - NoC(E,F,G)= ?



Coupling between object classes (CBO)



- For a class, C, the CBO metric is the number of other classes to which the class is coupled
- A class, X, is coupled to class C if
  - X operates on (affects) C or
  - C operates on X
- Excessive coupling indicates weakness of class encapsulation and may inhibit reuse
- High coupling also indicates that more faults may be introduced due to inter-class activities

The CBO value is arrived at by getting ratio of the number of links to the number of classes

Smaller values are better

DePaul University

Response For Class (RFC)

- Mc_i # of methods M_i called in response to a message received by any object of a class c.
- Number of Distinct Methods and Constructors invoked by a Class
- This set includes the methods in the class, inheritance hierarchy, and methods that can be invoked on other objects

$$\sum_{i=1}^{n} Mc_i$$

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

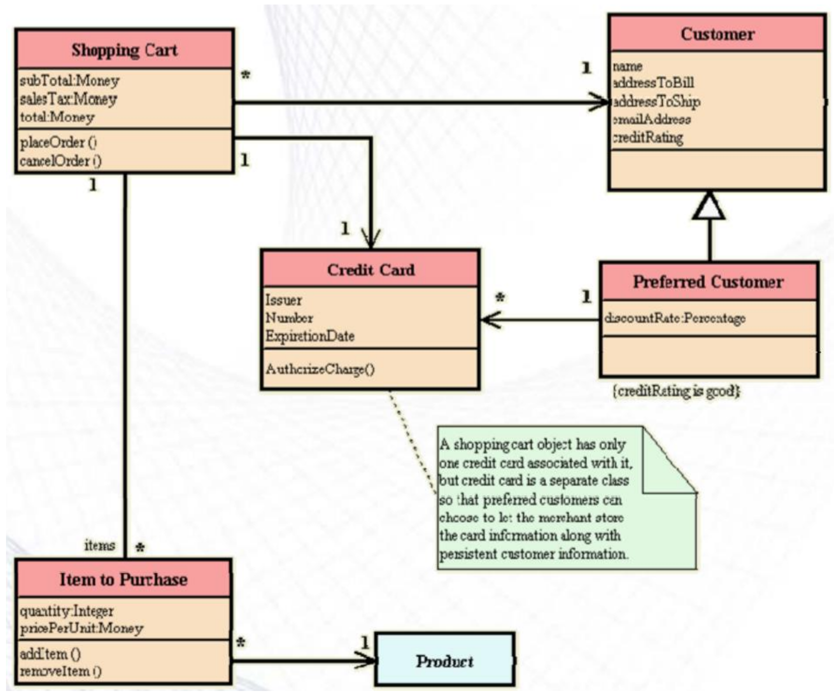
Response For Class (RFC) Example

```
Class C1{
M1(){…};
M2 (){…};
M3(){… C2.M1()};}

Class C2{
M1(){…};}
```

RFC (C1) = ?

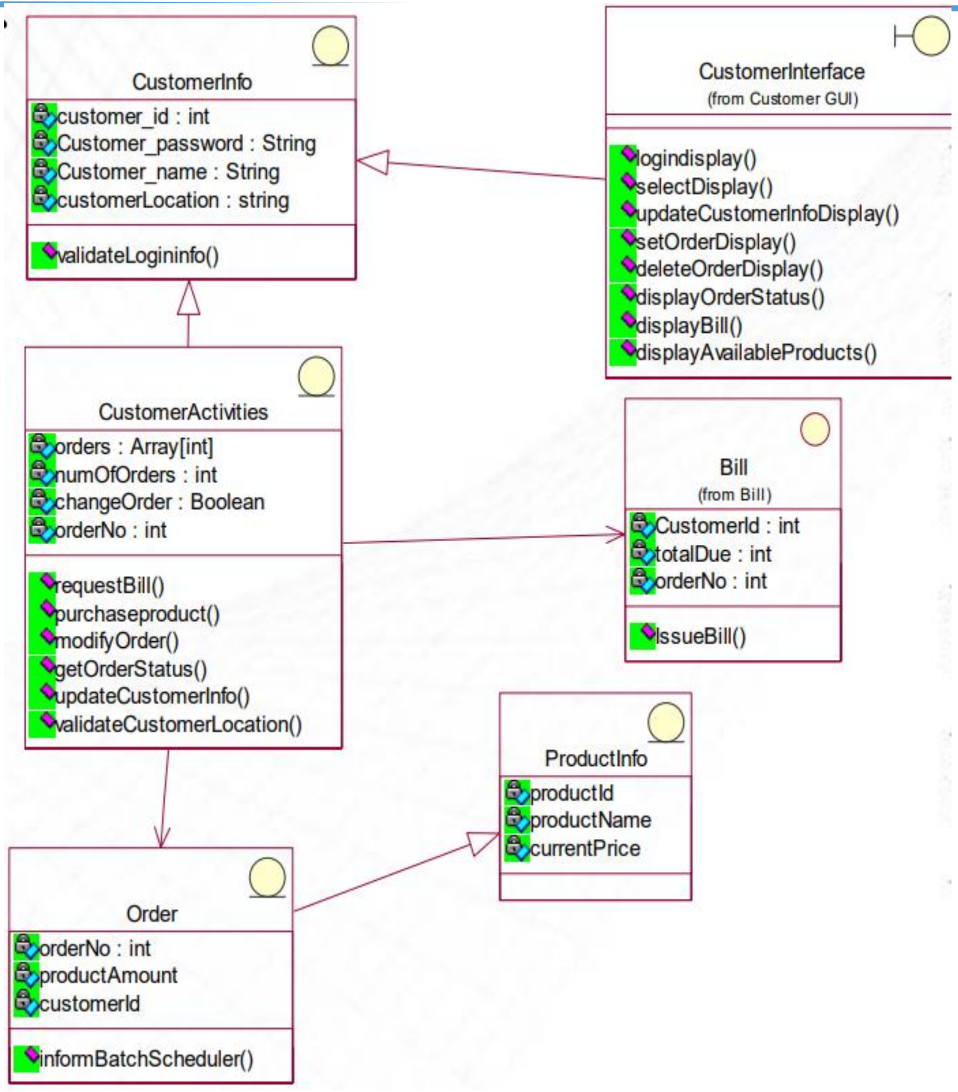
Response For Class (RFC) Example 2

- RFC for Preferred_Customer = ?

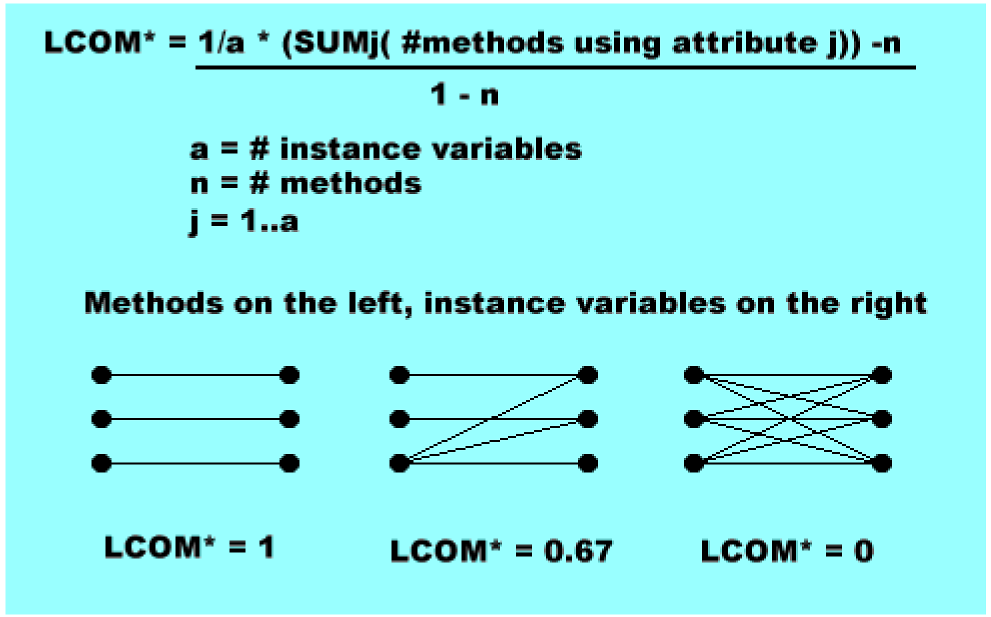


Response For Class (RFC) Example 3

- Determine the value of.
  - (1) “Average method per class”;
  - (2) “Response for a Class (RFC)”
    - CustomerActivities = ?
    - CustomerInterface = ?



Lack of cohesion metric (LCOM)



$$LCOM^* = \frac{1/a * (SUM_j(\#methods\ using\ attribute\ j)) - n}{1 - n}$$

a = # instance variables
n = # methods
j = 1..a

Methods on the left, instance variables on the right

Cohesiveness of methods within a class is desirable, since it promotes encapsulation

- the number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field. When this value is negative, the metric value is set to 0
  - LCOM* normalized version, range of values between 0..1
  - LCOM* = 0 if every method uses all instance variables
  - LCOM* = 1 if every method uses only one instance variable
- The larger the number of similar methods in a class the more cohesive the class is

Smaller values are better

Example #1:

$$\begin{tikzcd}
\underline{\text{Attributes}} & \underline{\text{Methods}} \\
a1 & m1 \arrow[l] \\
a2 & m2 \arrow[l] \arrow[ul] \\
a3 & m3 \arrow[l] \\
 & m4 \arrow[ul]
\end{tikzcd}$$

Example #2:

$$\begin{tikzcd}
\underline{\text{Attributes}} & \underline{\text{Methods}} \\
a1 & m1 \arrow[l] \\
a2 & m2 \arrow[l] \\
a3 & m3 \arrow[l] \\
 & m4
\end{tikzcd}$$

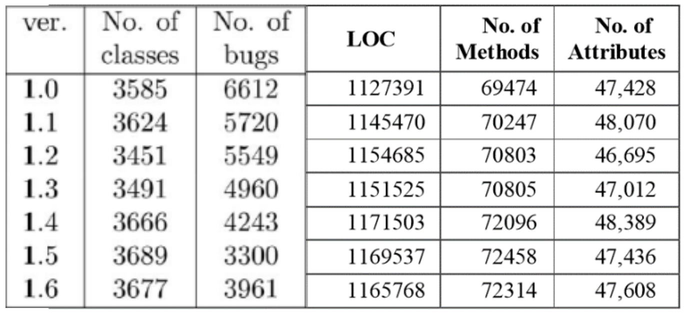
Lack of cohesion metric (LCOM)

- Problem with LCOM:
  - Classes with ”getters & setters” (getProperty(), setProperty()) get high LCOM values although this is not an indication of a problem
- Many versions that improves this metric (LCOM2, LCOM3, LCOM4…)
    - Henderson-Sellers
    - Total Correlation
    - Pairwise Field Irrelation
    - Etc.

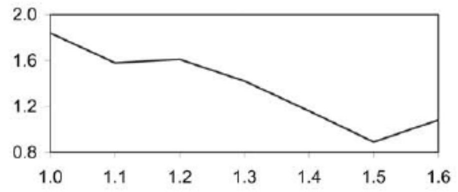
Example of metrics usage

SIZE METRICS FOR THE ANALYZED VERSION OF MOZILLA

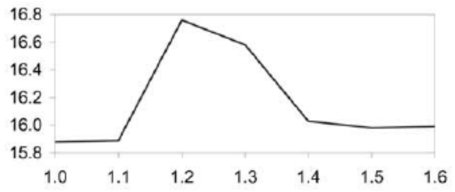
2002-2004



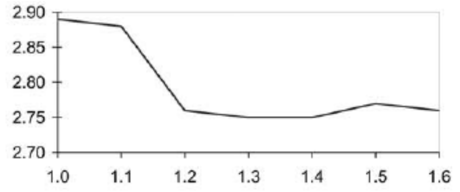
ver.	No. of classes	No. of bugs	LOC	No. of Methods	No. of Attributes
1.0	3585	6612	1127391	69474	47,428
1.1	3624	5720	1145470	70247	48,070
1.2	3451	5549	1154685	70803	46,695
1.3	3491	4960	1151525	70805	47,012
1.4	3666	4243	1171503	72096	48,389
1.5	3689	3300	1169537	72458	47,436
1.6	3677	3961	1165768	72314	47,608



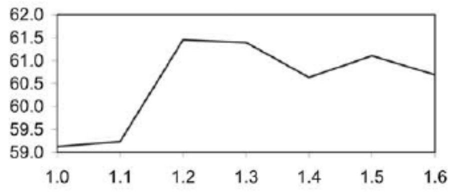
Bugs per Class



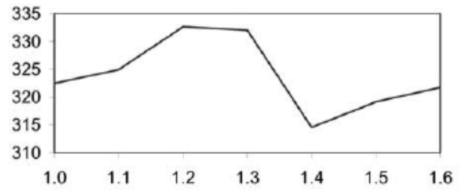
WMC



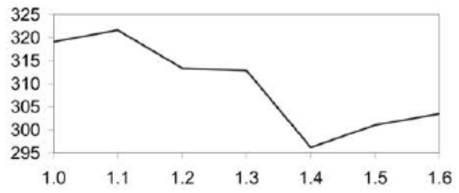
DIT



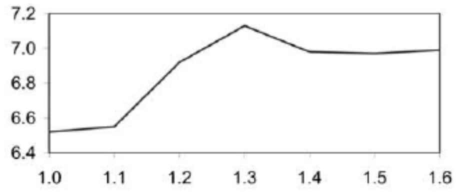
RFC



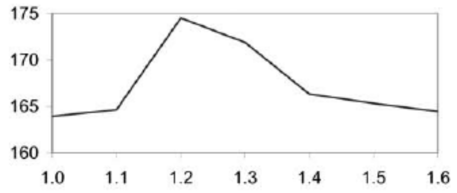
LCOM



LCOMN

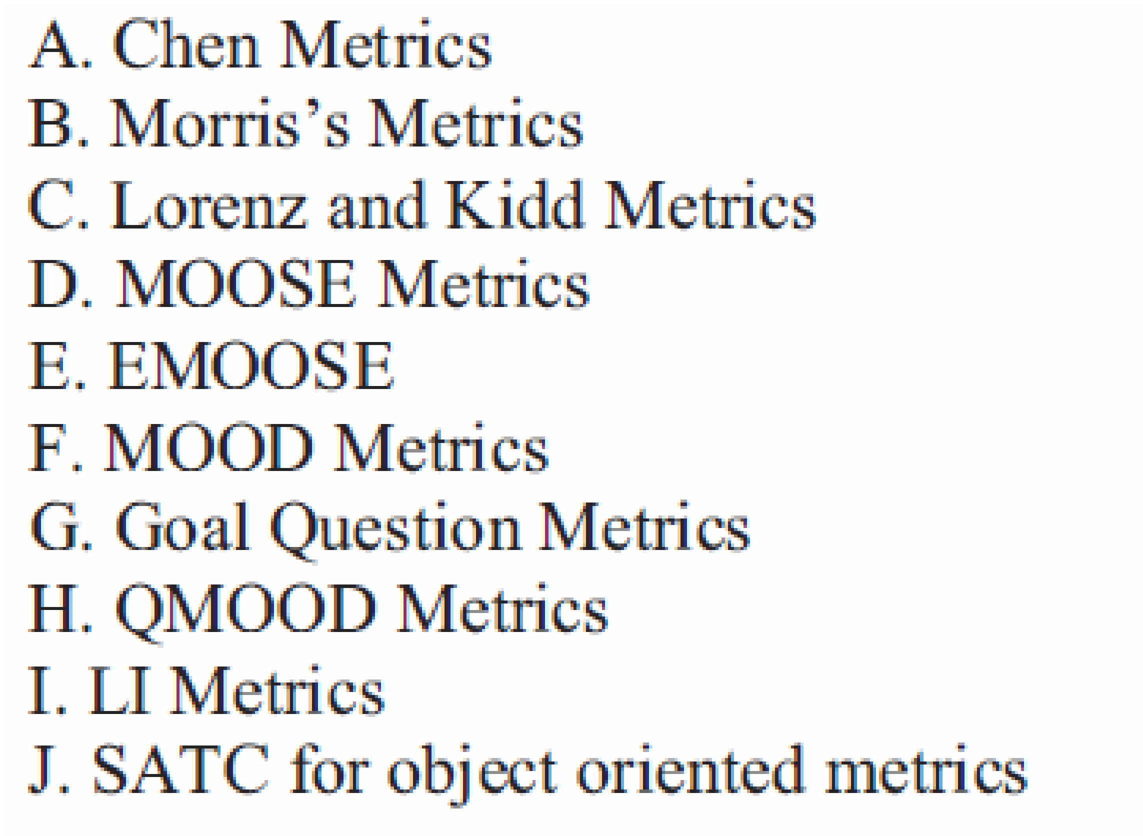


CBO



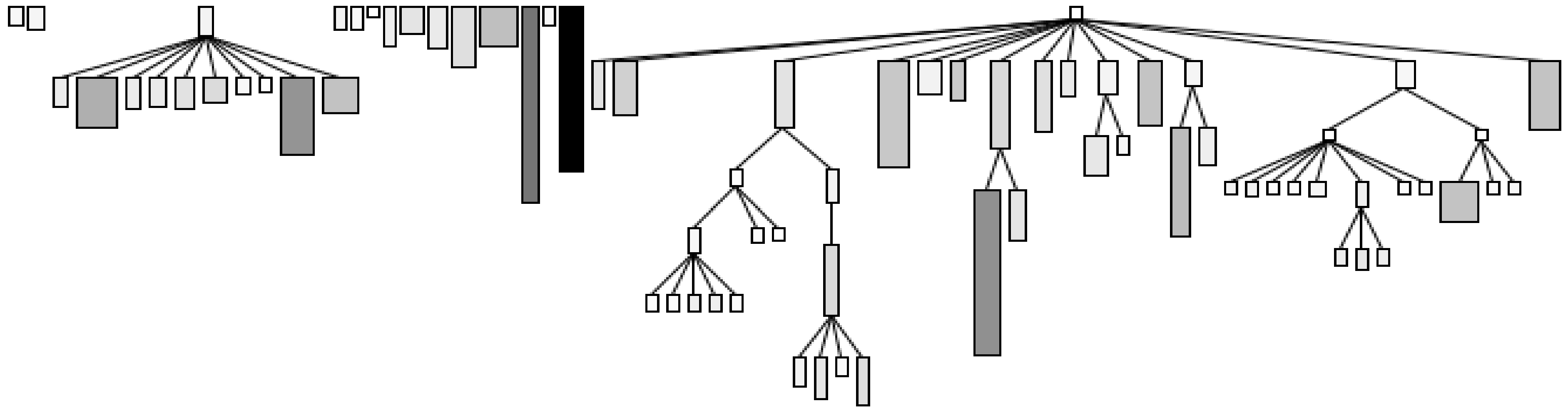
LOC per Class

Other metrics on the literature



A. Chen Metrics
B. Morris's Metrics
C. Lorenz and Kidd Metrics
D. MOOSE Metrics
E. EMOOSE
F. MOOD Metrics
G. Goal Question Metrics
H. QMOOD Metrics
I. LI Metrics
J. SATC for object oriented metrics

Visualization tools for Software Metrics

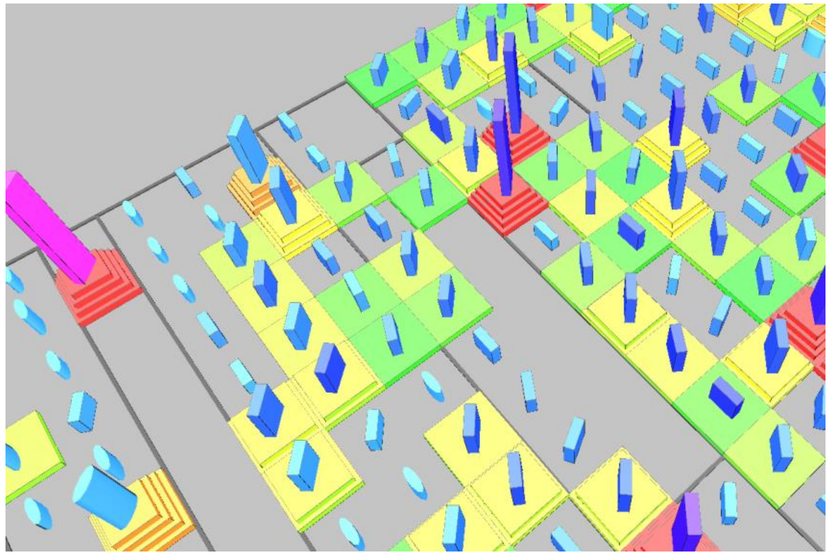


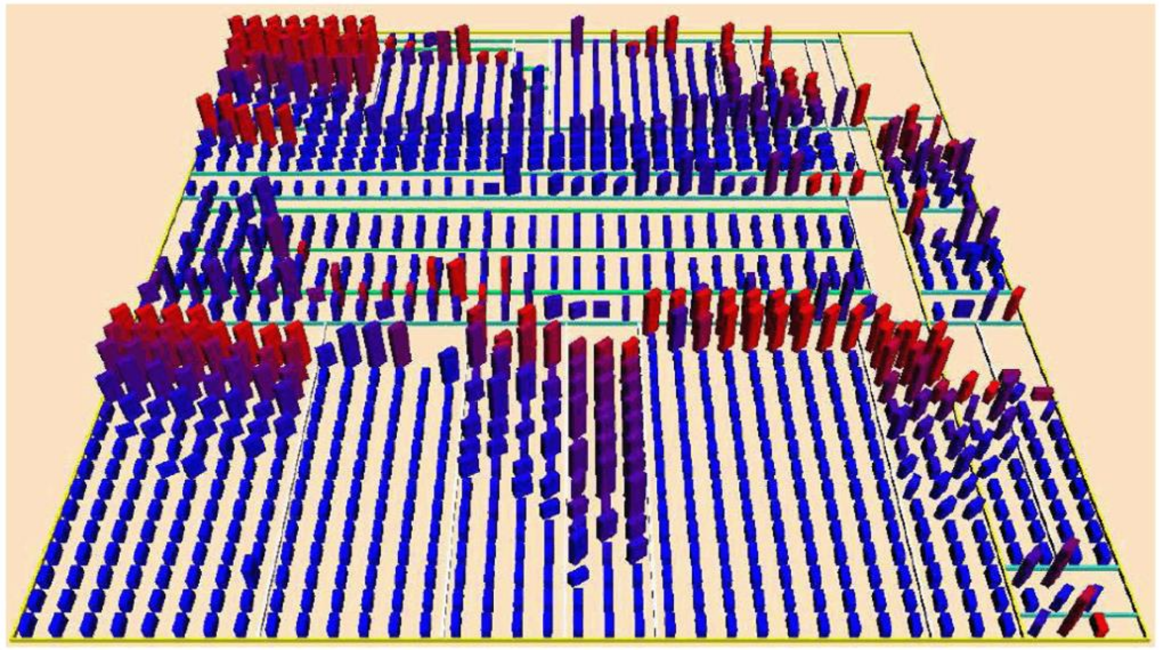
System Complexity View

Nodes =	Classes
Edges =	Inheritance Relationships

Width =	Number of Attributes
Height =	Number of Methods
Color =	Number of Lines of Code

Visualization tools for Software Metrics





Discussion :

Assignment 1 Description