CSC 2210
Object Oriented Analysis & Design

Dr. Akinul Islam Jony

Associate Professor

Department of Computer Science, FSIT

American International University - Bangladesh (AIUB)

akinul@aiub.edu

OO Software Metrics

>> Object-Oriented Concepts
>> OO Software Metrics vs. Conventional Software Metrics
>> OO Metrics Suite

Object-Oriented Concepts

Object-Orientation programming is a methodology for system analysis, design and implementation that supports integration of functional and data-oriented programming and system development.

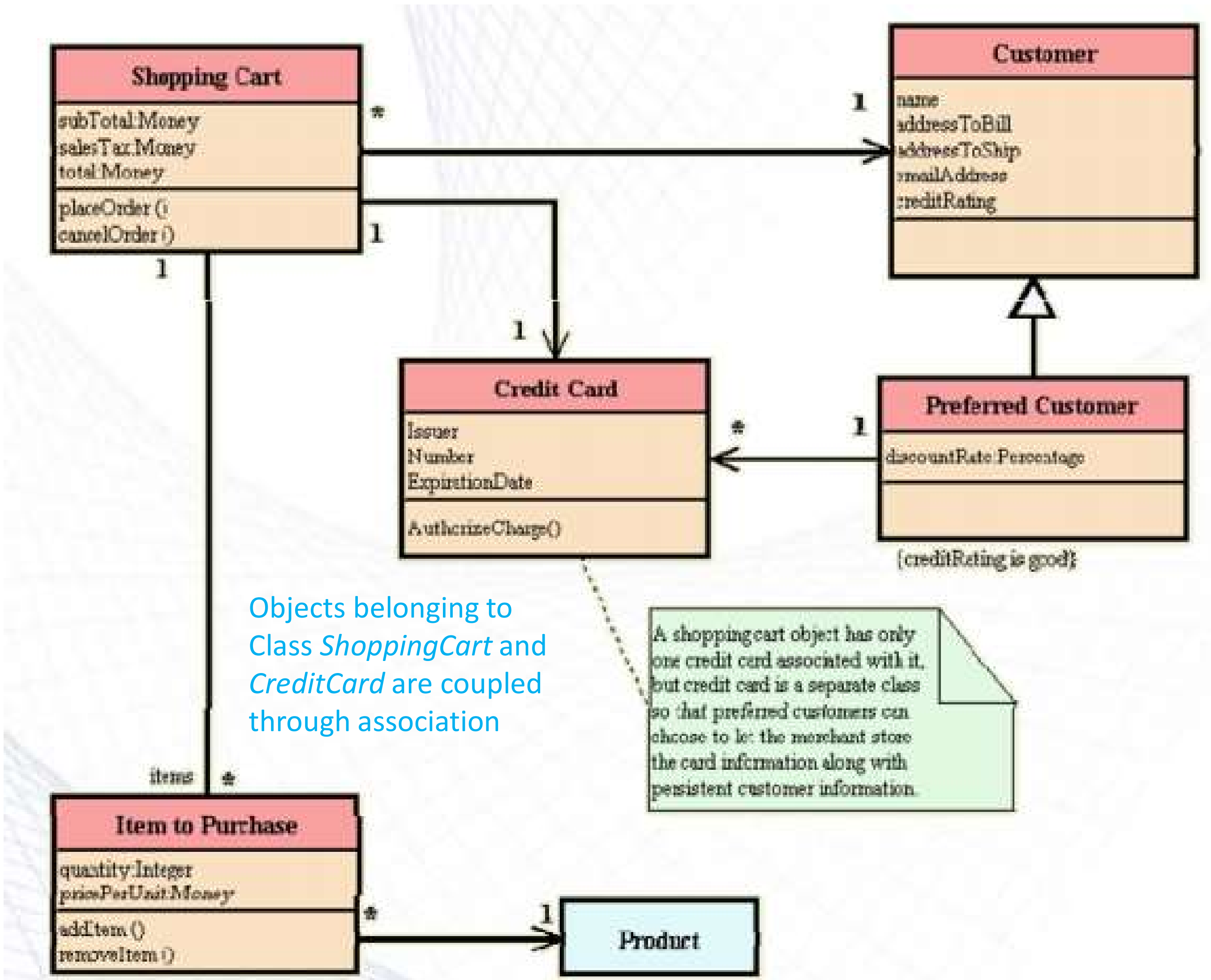
Object-Oriented Concepts

Terminology	Description
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 can be requested from any object of the class to affect behavior.

Object-Oriented Concepts

Terminology	Description
Package	A package is 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-Oriented Concepts



Objects belonging to Class *ShoppingCart* and *CreditCard* are coupled through association

Class *PreferredCustomer* is a child of Class *Customer* and inherits attributes and methods from *Customer*

Class Diagram: Electronic Shopping Cart

OO Software Metrics vs. Conventional Software Metrics

OO Software Metrics are different because of:

→ Localization
→Encapsulation
→ Information hiding
→ Inheritance
→ Reuse

Localization

→ Localization means placing items in close (physical) proximity (*property of being close together*) to each other.

- Functional decomposition (localize information around functions).
- Data localization (localize information around data).
- Object-oriented approaches (localize information around objects).

→ In conventional software, localization is based on functionality. Therefore:

- Metrics gathering has traditionally focused on functions and functionality
- Units of software were set to be functional, thus metrics focusing on component relationships emphasized functional interrelationships, e.g., module coupling and cohesion.

→ In object-oriented software, however, localization is based on **objects**. This means:

- Metrics identification and gathering effort must recognize the "object" as the basic unit of software.
- Within systems of objects, the localization between functionality and objects is not a one-to-one relationship.
- For example, one function may involve several objects, and one object may provide many functions.

Encapsulation

→ Encapsulation is the packaging (or binding together) of a collection of items:

- Low-level examples of encapsulation include records and arrays, procedures, subroutines.
- OO programming languages allow higher-level encapsulating, e.g., classes in C++ and Java.

→ Encapsulation has two major impacts on metrics:

- The basic unit will no longer be the program, but rather the object.
- We have to modify our thinking on characterizing and estimating systems.

Information Hiding

→ Information hiding is the suppression (or hiding) of details.

→ The general idea is that we show only that information which is necessary to accomplish our immediate goals.

→ There are degrees of information hiding ranging from partially restricted visibility (e.g., public or private operations) to total invisibility (e.g., subsystems).

→ Encapsulation and information hiding may not be the same thing, e.g., an item can be encapsulated but may still be totally visible (e.g., a package).

→ Information hiding plays a direct role in such metrics as object coupling and the degree of information hiding.

Inheritance

→ Inheritance is a mechanism whereby one object acquires characteristics from one, or more, other objects.

- Some OO languages support single inheritance (e.g., Java), some support multiple inheritance (e.g., C++).

→ Many OO software engineering metrics are based on inheritance, e.g.:

- Number of children (number of immediate specializations)
- Number of parents (number of immediate generalizations)
- Class hierarchy nesting level (depth of a class in an inheritance hierarchy)

Reuse

→ In OO development, reuse is a central issue

- Reuse of libraries or frameworks
- Reuse through inheritance
- Meta-code level reuse:
  - -- Patterns
  - -- Business objects

→ Reuse changes development process

- Build reusable components
- Find and reuse components

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

<u>What we want to measure for a package?</u>

→ Number of Classes, Operations (Methods), Attributes (Variables) Attributes (Variables) (Averaged 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?

→ Instance variables
→ 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

OO Metrics Suite

→ Object Oriented Metrics seek to measure the unique attributes of Object Oriented design as opposed to software developed using other methods.

→Chidamber and Kemerer felt that software metrics developed with traditional methods in mind did not readily lend themselves to Object Oriented notions such as classes, inheritance, encapsulation and message passing.

→ They proposed 6 metrics unique to Object Oriented systems. These metrics measure various attributes including size and complexity and are constructed with a strong degree of theoretical and mathematical rigor.

OO Metrics Suite

METRIC	OO CONSTRUCT
Weighted Methods per Class (WMC)	Class/Operation
Response For a Class (RFC)	Class/Operation
Lack of Cohesion (LCOM)	Class/Operation
Coupling Between Objects (CBO)	Coupling
Depth of Inheritance Tree (DIT)	Inheritance
Number of Children (NoC)	Inheritance

Weighted Methods per Class (WMC)

→ WMC is a metric of size and complexity. It is defined as: **$WMC = \Sigma c_i$**

- where c_i is the complexity of each different method $c_1, c_2, \ldots c_n$ in class C.
- If each method were assigned a complexity of 1 then the WMC for class C would equal the number of methods in the class.

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

→ The larger the number of methods in a class, the greater the potential impact on children; children inherit all of the methods defined in the parent class.

→ Classes with large numbers of methods are likely to be more application specific, limiting the possibility of reuse.

→ Example: WMC is calculated by counting the number of methods in each class, therefore:

- WMC for *ShoppingCart* = 2
- WMC for *CreditCard* = 1

Depth of Inheritance Tree (DIT)

→ The depth of a class within the inheritance hierarchy is the maximum number of steps from the class node to the root of the tree and is measured by the number of ancestor classes.

→ The deeper a class is within the hierarchy, the greater the number of methods it is likely to inherit making it more complex to predict its behavior.

→ Deeper trees constitute greater design complexity, since more methods and classes are involved, but the greater the potential for reuse of inherited methods.

→ Example:

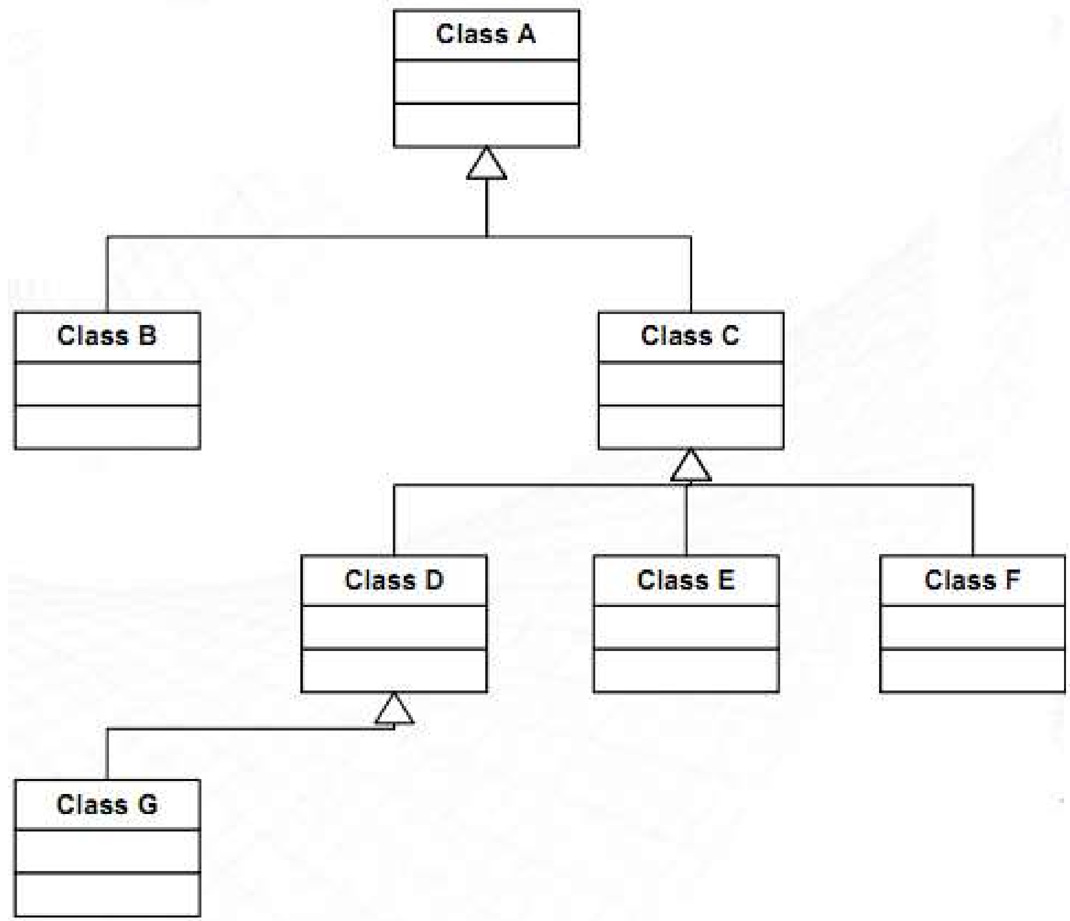
-*Customer* is the root and has a DIT of 0.

-The DIT for *PreferredCustomer* is 1.

Depth of Inheritance Tree (DIT)

Another Example

- DIT(A) = 0
- DIT(B, C) = 1
- DIT(D,E,F) = 2
- DIT(G) = 3



Number of Children (NoC)

→ The number of children is the number of immediate subclasses subordinate to a class in the hierarchy.

→ It is an indicator of the potential influence a class can have on the design and on the system.

→ The greater the NoC, gives an idea of the potential influence a parent class has on design. If a class has a large number of sub-classes, it could require more testing of its methods.

→ The greater the NoC, the greater the reuse since inheritance is a form of reuse.

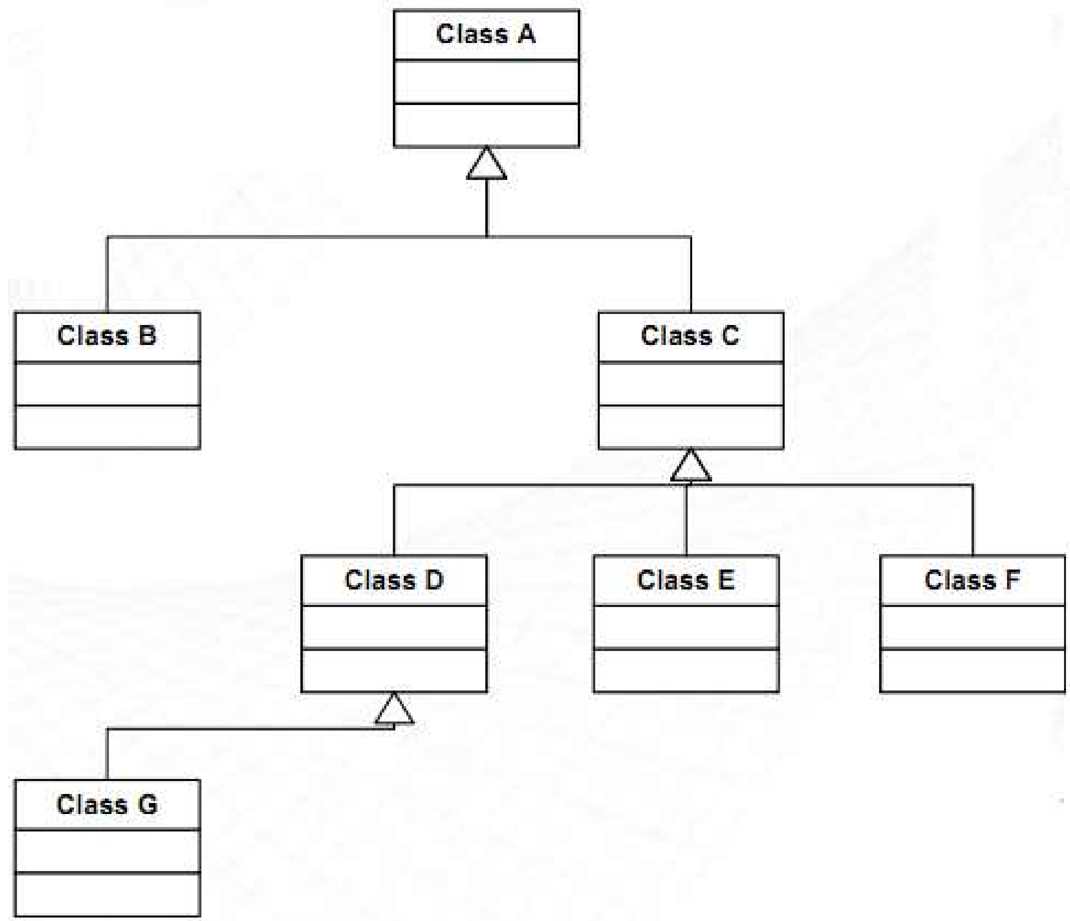
→ Example:

- *Customer* has an NOC of 2.
- NOC for *PreferredCustomer* is 0 since it is a terminating or leaf node in the tree structure.

Number of Children (NoC)

Another Example

- NoC(A) = 2
- NoC(C) = 3
- NoC(D) = 1
- NoC(B, E, F, G) = 0



Coupling Between Objects (CBO)

→ CBO metric therefore is a measure of non-inherited interactions between classes.

→ CBO is a count of the number of other classes to which a class is coupled.

→ It is measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends.

→ Excessive coupling is detrimental to modular design and prevents reuse.

→The more independent a class is, the easier it is reuse in another application.

→ Two classes may have excessive coupling (too many messages passing between them). This implies that those classes should be combined into one class.

Response for a Class (RFC)

→ RFC is the count of the set of all methods that can be invoked in response to a message to an object of the class or by some method in the class.

→This metric looks at combination of the complexity of a class through the number of methods and the amount of communication with other classes.

→ The larger the number of methods that can be invoked from a class through messages, the greater the complexity of the class.

→ Example:

- RFC for *PreferredCustomer* = 0 (self) + 0 (Customer) + 1 (*CreditCard*) = 1

Lack of Cohesion Methods (LCOM)

→ Lack of Cohesion (LCOM) measures the dissimilarity of methods in a class by instance variable or attributes.

→ If a class has different methods performing different operations on the same set of instance variables, the class has cohesion.

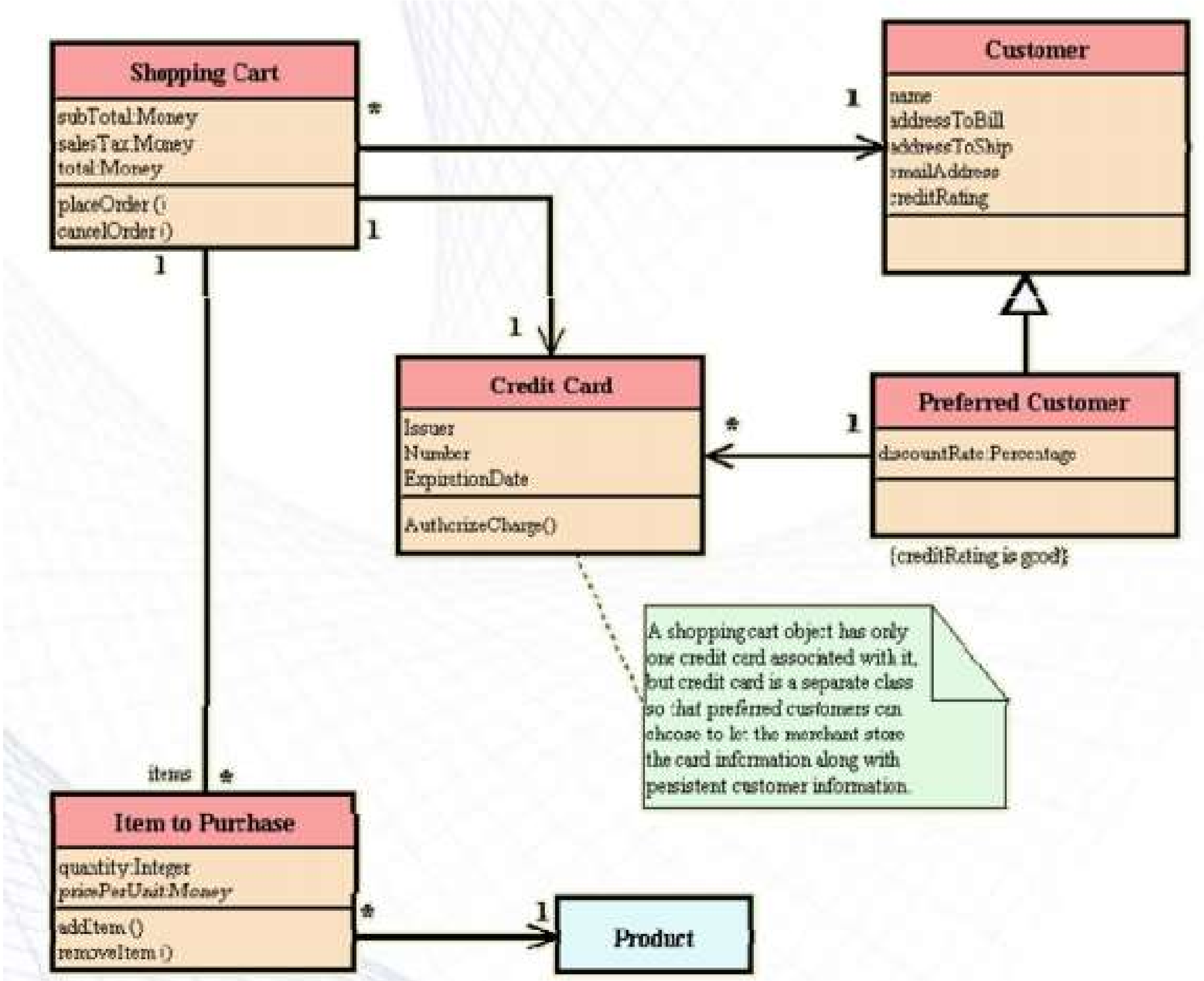
→ A highly cohesive module should stand alone; high cohesion indicates good class subdivision.

→ Lack of cohesion or low cohesion increases complexity, thereby increasing the likelihood of errors during the development process.

→ High cohesion implies simplicity and high reusability.

Lack of Cohesion Methods (LCOM)

Example:
Alternative design with high LCOM: Assume that the *CreditCard* and *PreferredCustomer* classes are merged. There will be relatively few common attributes and methods among the objects that may belong to this class.



Example

Calculate: LCOM

CLASS A
a1 a2 a3 a4
A1(a1, a2) A2(a1) A3 (a4) A4 (a1, a4)

LCOM = |P|-|Q|, if |P| > |Q|, otherwise 0

Pairs:
(A1, A2), (A1, A3), (A1, A4), (A2, A3), (A2, A4), (A3, A4)

P = 2 (Non-Cohesive pairs)
Q = 4 (Cohesive pairs)

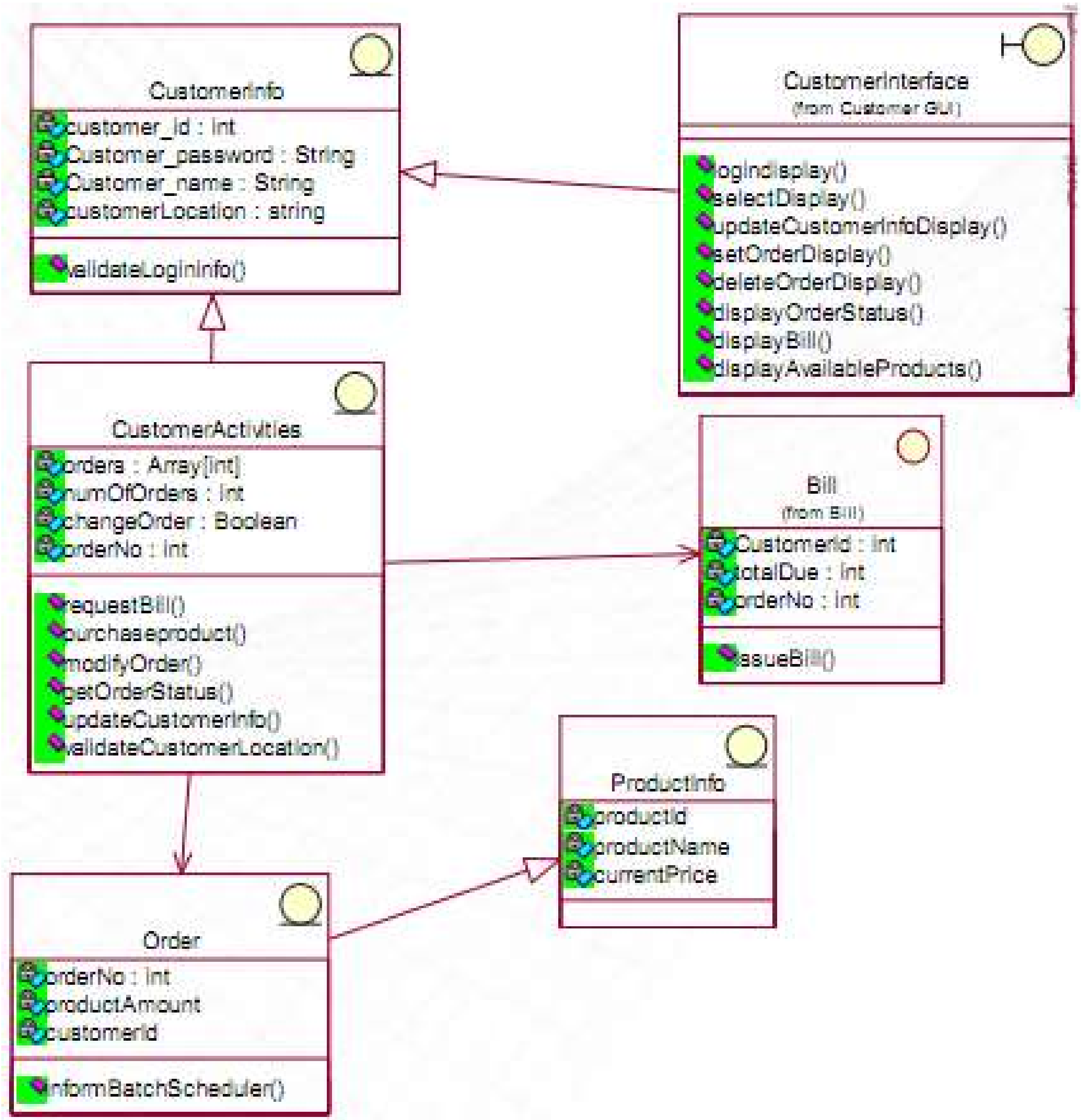
Q > P

LCOM = 0

Example

Determine the value of

(1) “Average method per class”;
(2) “Response for a Class (RFC)”
for *CustomerActivities* and *CustomerInterface*



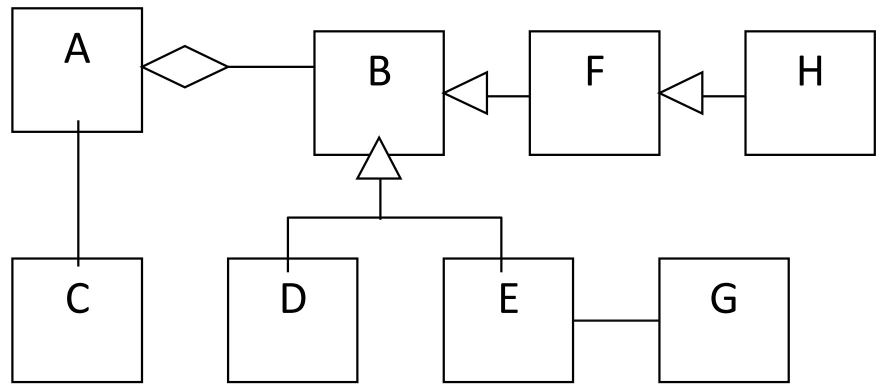
Average method per class
= 17/6 = 2.83

For CustomerActivities RFC
= 9

For CustomerInterface RFC
= 9

Example

Calculate: NOC, DIT, CBC



CLASS	NOC	Influence on Design	DIT	Reuse Potential	CBC
A	0	Low	0	Low	Low
B	3	Highest	0	Low	Highest
C	0	Low	0	Low	Lowest
D	0	Low	1	Moderate	Low
E	0	Low	1	Moderate	Moderate
F	1	Moderate	1	Moderate	Moderate
G	0	Low	0	Low	Lowest
H	0	Low	2	Highest	Lowest

Interpretation

METRICS	OBJECTIVE
Cyclomatic Complexity	Low
Lines of Code/Executable Statements	Low
Comment Percentage	~ 20 – 30 %
Weighted Methods per Class (WMC)	Low
Response for a Class (RFC)	Low
Lack of Cohesion of Methods (LCOM)	Low
Cohesion of Methods	High
Coupling Between Objects (CBO)	Low
Depth of Inheritance Tree (DIT)	Low (trade-off)
Number of Children (NoC)	Low (trade-off)

[Rosenberg, et al.]

References

→**Object Oriented Metrics**

Lecture slides by B.H. Far

Department of Electrical & Computer Engineering

University of Calgary