

Object Oriented Programming with C++

Introduction and Course details

Lecturer : Astha Sharma

Chapter 7

Object oriented design

Object oriented software development

Object Oriented Design



The diagram consists of seven diamond-shaped icons arranged in two rows. Each icon has a white circle in the center with a colored border. The top row contains four icons: 'Objects' (light blue), 'Classes' (green), 'Messages' (orange), and 'Abstraction' (dark blue). The bottom row contains three icons: 'Encapsulation' (red), 'Inheritance' (teal), and 'Polymorphism' (purple). The text is in a dark red, serif font.

Objects

Classes

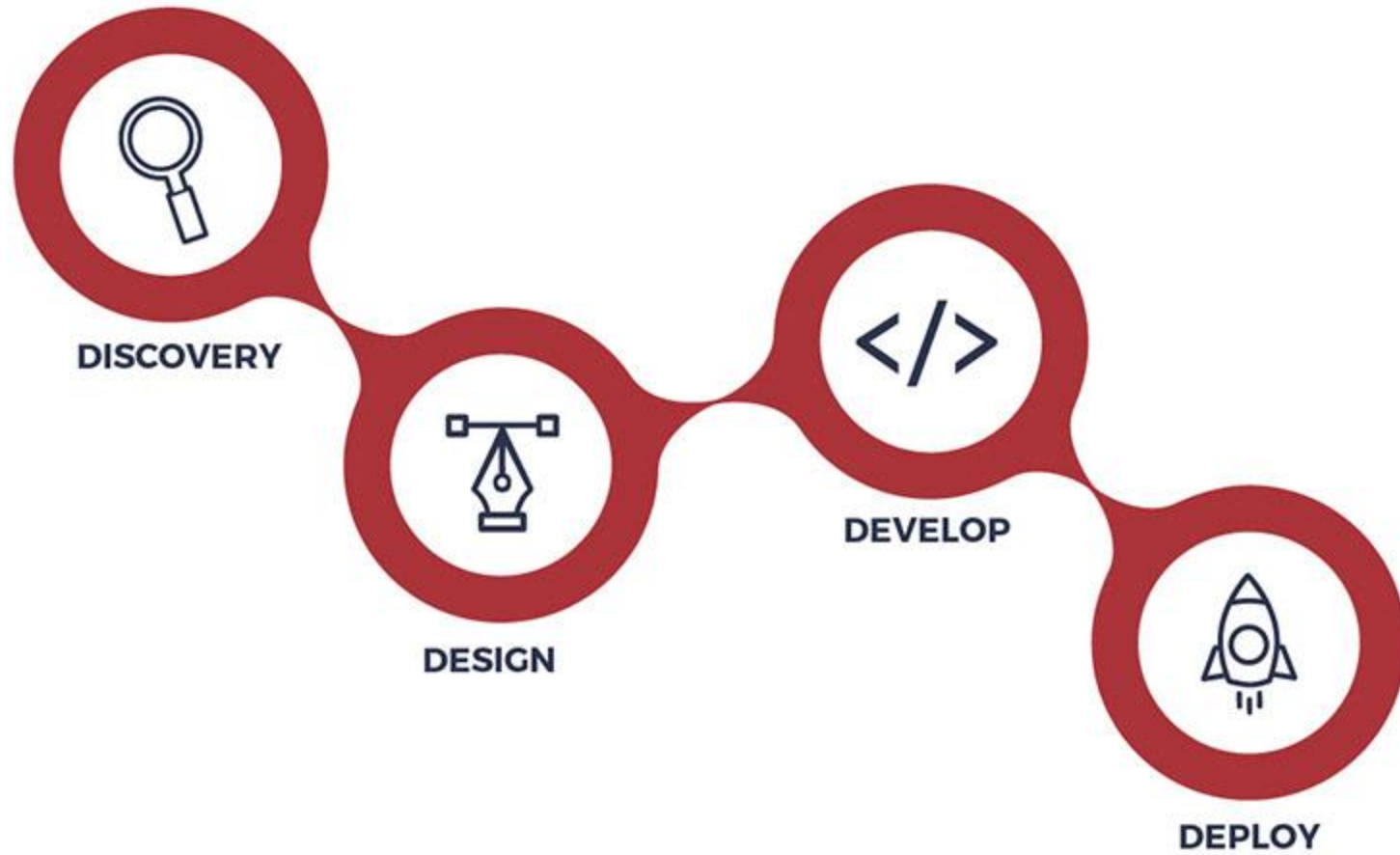
Messages

Abstraction

Encapsulation

Inheritance

Polymorphism



Object oriented system development

- Anything is easier to make when there is a clear design or thought.
- In the system development too, the process can be simplified by brainstorming and categorizing what objects or classes can be used for what reasons.
- It is more natural to traverse from analysis to implementation as it is capable of modeling real world problem very well.
- The object oriented paradigm is still under development as new challenges are arising every day.
- Software engineering is aimed to deal with various tools, methods or procedures required for controlling and managing the software development.

Seven Phases of Software Development Life Cycle

Planning



Design & Prototyping



Testing



Operations & Maintenance



Define Requirements




Software Development




Deployment




CRC CARDS

- 
- A Class Responsibility Collaborator (CRC) model (Beck & Cunningham 1989) is a collection of standard index cards that have been divided into three sections, as depicted in Figure 1.
 - A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.
 - Figure 2 presents an example of two hand-drawn CRC cards.



Class Name	
Responsibilities	Collaborators

Customer			Order		
Places	orders				
Knows	name				
Knows	address				
Knows	customer number				
Knows	order history				
Order			Order Item		
Knows	placement	date			
Knows	delivery	date			
Knows	total				
Knows	appl. cable	taxes			
Knows	order	number			
Knows	order	items			

- 
- Although CRC cards were originally introduced as a technique for teaching object-oriented concepts, they have also been successfully used as a full-fledged modeling technique.
 - A class represents a collection of similar objects.
 - An object is a person, place, thing, event, or concept that is relevant to the system at hand.
 - For example, in a university system, classes would represent students, tenured professors, and seminars.


NAME

- The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as *Student*, *Professor*, and *Seminar*.
- You use singular names because each class represents a generalized version of a singular object.
- Although there may be the student John O'Brien, you would model the class *Student*.

- The information about a student describes a single person, not a group of people.
- Therefore, it makes sense to use the name *Student* and not *Students*.
- Class names should also be simple.
- For example, which name is better: *Student* or *Person who takes seminars*?

RESPONSIBILITY

- A responsibility is anything that a class knows or does.
- For example, students have names, addresses, and phone numbers.
- These are the things a student knows.
- Students also enroll in seminars, drop seminars, and request transcripts.
- These are the things a student does.

- 
- The things a class knows and does constitute its responsibilities.
 - Important: A class is able to change the values of the things it knows, but it is unable to change the values of what other classes know.

COLLABORATORS

- Sometimes a class has a responsibility to fulfill, but not have enough information to do it.
- For example, as you see in Figure 3 students enroll in seminars.
- To do this, a student needs to know if a spot is available in the seminar and, if so, he then needs to be added to the seminar.
- However, students only have information about themselves (their names and so forth), and not about seminars.




- 
- What the student needs to do is collaborate/interact with the card labeled *Seminar* to sign up for a seminar.
 - Therefore, *Seminar* is included in the list of collaborators of *Student*.

FIGURE 3: CRC of student

Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

- 
- Collaboration takes one of two forms: A request for information or a request to do something.
 - For example, the card *Student* requests an indication from the card *Seminar* whether a space is available, a request for information.
 - *Student* then requests to be added to the *Seminar*, a request to do something.

- 
- Another way to perform this logic, however, would have been to have *Student* simply request *Seminar* to enroll himself into itself.
 - Then have *Seminar* do the work of determining if a seat is available and, if so, then enrolling the student and, if not, then informing the student that he was not enrolled.



So how do you create CRC models? Iteratively perform the following steps:

1. **Find classes.**
 - Finding classes is fundamentally an analysis task because it deals with identifying the building blocks for your application.
 - A good rule of thumb is that you should look for the three-to-five main classes right away, such as *Student*, *Seminar*, and *Professor* in Figure 4.




2. Find responsibilities.

- You should ask yourself what a class does as well as what information you wish to maintain about it.
- You will often identify a responsibility for a class to fulfill a collaboration with another class.


3. Define collaborators.

- A class often does not have sufficient information to fulfill its responsibilities.
- Therefore, it must collaborate (work) with other classes to get the job done.
- Collaboration will be in one of two forms: a request for information or a request to perform a task.
- To identify the collaborators of a class for each responsibility ask yourself "does the class have the ability to fulfill this responsibility?"

- 
- If not then look for a class that either has the ability to fulfill the missing functionality or the class which should fulfill it.
 - In doing so you'll often discover the need for new responsibilities in other classes and maybe even the need for a new class or two.


4. Move the cards around.


- To improve everyone's understanding of the system, the cards should be placed on the table in an intelligent manner.
- Two cards that collaborate with one another should be placed close together on the table, whereas two cards that don't collaborate should be placed far apart.

- 
- Furthermore, the more two cards collaborate, the closer they should be on the desk.
 - By having cards that collaborate with one another close together, it's easier to understand the relationships between classes.



UML

- 
- **UML**, short for Unified Modeling Language, is a standardized modeling language consisting of an integrated set of diagrams.
 - It is developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems.
 - It is also used for business modeling and other non-software systems.

- 
- The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
 - The UML is a very important part of developing object oriented software and the software development process.
 - The UML uses mostly graphical notations to express the design of software projects.
 - Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.



IMPORTANCE OF UML:

1. It helps us to visualize a system.
2. It enable us to specify structure and behaviour of a system.
3. It provides a template to make the construction of a system.
4. It provides a means of documentation.

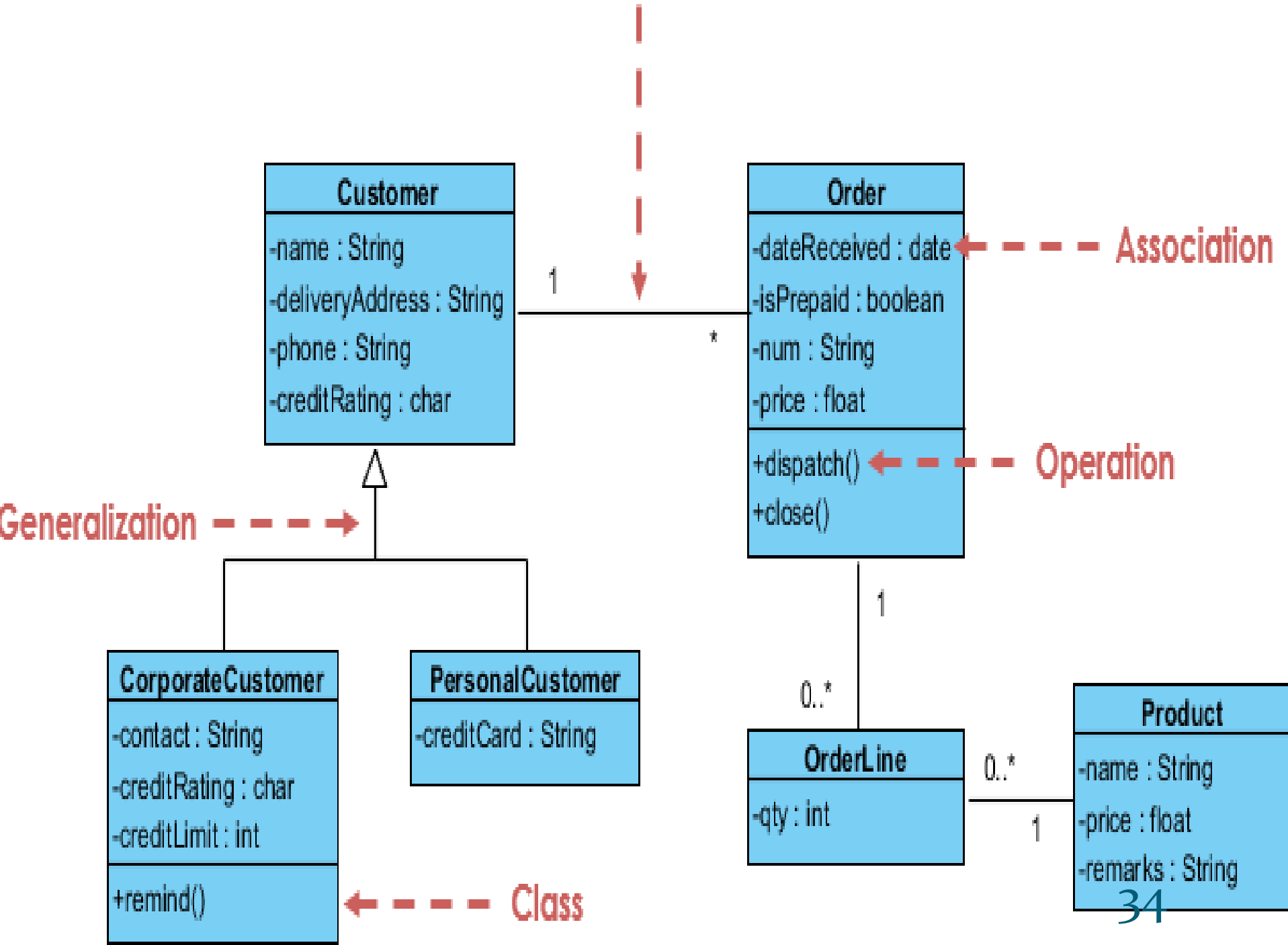
Class Diagram

- The class diagram is a central modeling technique that runs through nearly all object-oriented methods.
- This diagram describes the types of objects in the system and various kinds of static relationships which exist between them.

Relationships between classes

There are three principal kinds of relationships which are important:

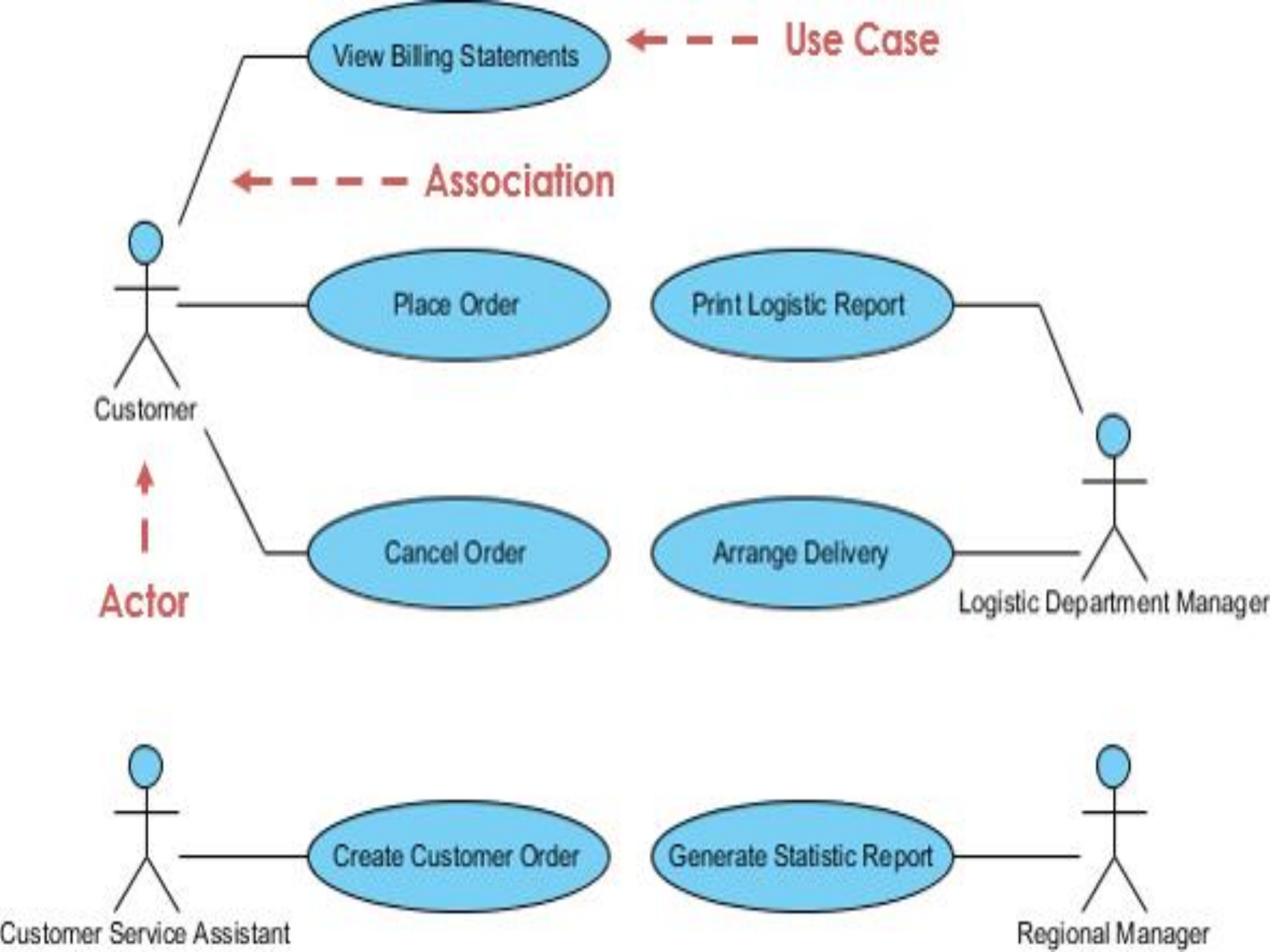
1. **Association** - represent relationships between instances of types (a person works for a company, a company has a number of offices).
2. **Inheritance** - the most obvious addition to ER diagrams for use in OO. It has an immediate correspondence to inheritance in OO design.
3. **Aggregation** - Aggregation, a form of object composition in object-oriented design.



Use Case Diagram


- A use-case model describes a system's functional requirements in terms of use cases.
- It is a model of the system's intended functionality (use cases) and its environment (actors).
- Use cases enable you to relate what you need from a system to how the system delivers on those needs.
- Think of a use-case model as a menu, much like the menu you'd find in a restaurant.
- By looking at the menu, you know what's available to you, the individual dishes as well as their prices.

- You also know what kind of cuisine the restaurant serves: Italian, Mexican, Chinese, and so on.
- By looking at the menu, you get an overall impression of the dining experience that awaits you in that restaurant.
- The menu, in effect, "models" the restaurant's behavior.
- Because it is a very powerful planning instrument, the use-case model is generally used in all phases of the development cycle by all team members.



Sequence Diagram

- The Sequence Diagram models the collaboration of objects based on a time sequence.
- It shows how the objects interact with others in a particular scenario of a use case.
- With the advanced visual modeling capability, you can create complex sequence diagram in few clicks.



- Besides, some modeling tool such as Visual Paradigm can generate sequence diagram from the flow of events which you have defined in the use case description.

User

Theater :
Theater 2

Server : Server
1

Insert Card

Select date

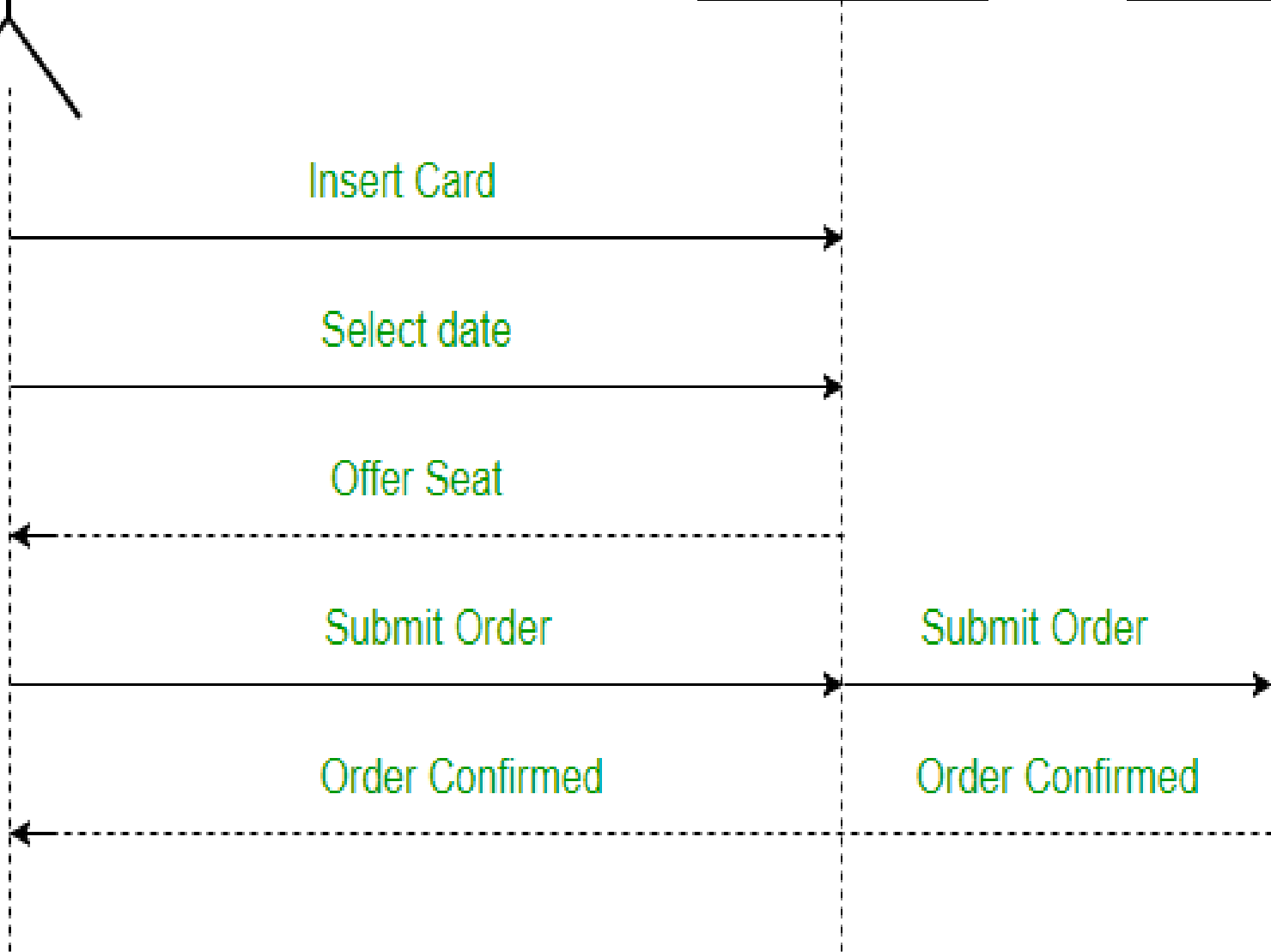
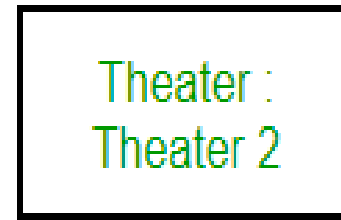
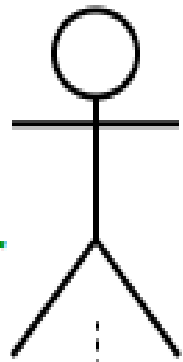
Offer Seat

Submit Order

Order Confirmed

Submit Order

Order Confirmed



Programming in small and large



Programming in small

Characteristics:

- Code is developed by single programmer or maybe a very small group of programmers.
- A single individual can understand all aspects of the project from top to bottom, beginning to end.
- The major problem in s/w development lifecycle is the design with the problem at hand.

Programming in large


Characteristics:


- Code is developed by large team of programmers.
- Individuals involved in the specification or design of the system may differ from those involved in integration of various components in final product.
- No single individual can be considered responsible for the entire project and also does not necessarily need to understand all the aspects of the project.
- The major problem in s/w development lifecycle is the management of details and the communication of information between diverse portions of the project.

Responsibility Driven Design


Characteristics:

- **Responsibility-driven design** is a design technique in Object-oriented programming (OOP), which improves encapsulation by using the client-server model.
- It focuses on the contract by considering the actions that the object is responsible for and the information that the object shares.
- It was proposed by Rebecca Wirfs-Brock and Brian Wilkerson.
- Responsibility-driven design is in direct contrast with data-driven design, which promotes defining the behavior of a class along with the data that it holds.

- 
- Data-driven design is not the same as data-driven programming which is concerned with using data to determine the control flow not class design.
 - In the client-server model they refer to, both the client and the server are classes or instances of classes.
 - At any particular time, either the client or the server represents an object.
 - Both the parties commit to a contract and exchange information by adhering to it.

- 
- The client can only make the requests specified in the contract and the server must answer these requests.
 - Thus, responsibility-driven design tries to avoid dealing with details, such as the way in which requests are carried out, by instead only specifying the intent of a certain request.
 - The benefit is increased encapsulation, since the specification of the exact way in which a request is carried out is private to the server.

Software components

- 
- An abstract design entity with which we can associate responsibilities for different tasks.
 - May eventually be a class function module.
 - A component must have a small, well-defined, understandable set of responsibility.
 - It should interact with other components as little as possible.

Behaviour

- The behaviour of the system is the set of actions it can perform.
- The complete description of all the behaviours for a component is sometimes called the protocol.
- For the recipe component this include activities such as editing the preparation instructions, displaying the instruction in terminal screen or printing the copy of the recipe.

State

- State of the components represent all the information of the components held within it.
- The state of recipe component are ingredients and set of instructions.
- Most components consist both the behaviour and state.

Instances and classes

- An individual representation of class is called instance(object).
- Instance variable is equal to data variable.
- Class is used to describe a set of objects with similar objects.

Coupling and cohesion

- **Cohesion** and **Coupling** deal with the quality of an OO design.
- Generally, good OO design should be loosely coupled and highly cohesive.
- Lot of the design principles, design patterns which have been created are based on the idea of “Loose coupling and high cohesion”.




- The aim of the design should be to make the application:

1. easier to develop
2. easier to maintain
3. easier to add new features
4. less Fragile.

Coupling:

- Coupling is the degree to which one class knows about another class.
- Let us consider two classes class **A** and class **B**.
- If class **A** knows class **B** through its interface only i.e it interacts with class **B** through its API then class **A** and class **B** are said to be loosely coupled.
- If on the other hand class **A** apart from interacting class **B** by means of its interface also interacts through the non-interface stuff of class **B** then they are said to be tightly coupled.

- 
- Suppose the developer changes the class **B**'s non-interface part i.e non API stuff then in case of loose coupling class **A** does not breakdown but tight coupling causes the class **A** to break.

So its always a good OO design principle to use loose coupling between the classes i.e all interactions between the objects in OO system should use the APIs.

- An aspect of good class and API design is that classes should be well encapsulated.

Cohesion:

- **Cohesion** is used to indicate the degree to which a class has a single, well-focused purpose.
- Coupling is all about how classes interact with each other, on the other hand cohesion focuses on how single class is designed.
- Higher the cohesiveness of the class, better is the OO design.



Benefits of Higher Cohesion:

1. Highly cohesive classes are much easier to maintain and less frequently changed.
2. Such classes are more usable than others as they are designed with a well-focused purpose.

Interface and implementation

- The interface view is the face seen by other programmers.
- It describes what a software component can perform.
- The implementation view is the face seen by the programmer working on a particular component.
- The difference between the implementation and interface is the important part of software component.