Unit 7: Object Oriented Design

Responsibility Implies Non-Interference

- Rather than syntactic aspects of a particular programming language like C++, the most important aspect of OOP is the design technique driven by the determination and delegation of responsibilities which is known as ***responsibility driven design (RDD)***. [Wirfs-Brock, 1989]
- When we make an object responsible for specific actions, we expect certain behaviors to be fulfilled by that object, at least when the rules are followed.
- Responsibility implies a degree of independence or non-interference.
- ***Example:*** If a manager of a company instructs an employee to prepare a report, the manager does not normally stand behind the employee and watch every step on how the report is being prepared. Instead, having issued a proper directive, the manager expects that the desired outcome will be produced.

- Difference between conventional programming and OOP is in many ways similar to the difference between actively supervising the employee to prepare the report versus determining and delegating the responsibility to the employee.
- In conventional programming, one portion of code in a software system is often closely tied to many other sections by control or data connections, like use of global variables, pointers or other inappropriate dependence.
- Responsibility Driven Design (RDD) attempts to cut or minimize such dependencies by focusing primarily on creating objects which are responsible for certain tasks rather than just encapsulating related data and functions together into classes or objects.

Programming in the Small and in the Large

Difference between the development of individual projects and of more sizeable projects is termed as programming in the small versus programming in the large.

A novice programmer will usually be acquainted only with programming in the small. However, many aspects of OOP are best understood as responses to problems encountered while programming in the large.

Programming in the Small

- Code developed by a single programmer or a very small team (only few people)
- A single individual (programmer) can understand all aspects of the project
- Major concern in the software development process is the design and development of algorithms for dealing with the problem being solved

Programming in the Large

- Software system developed by a large team
- Different individuals would be involved for different activities like:
  - specification or design of the system,
  - coding of individual components,
  - integration of various components,
  - testing or quality assurance
- No single individual can be considered responsible for the entire project
- The major concern in the software development process is the management of details and the communication of information between diverse portions of the project

Identification of Components

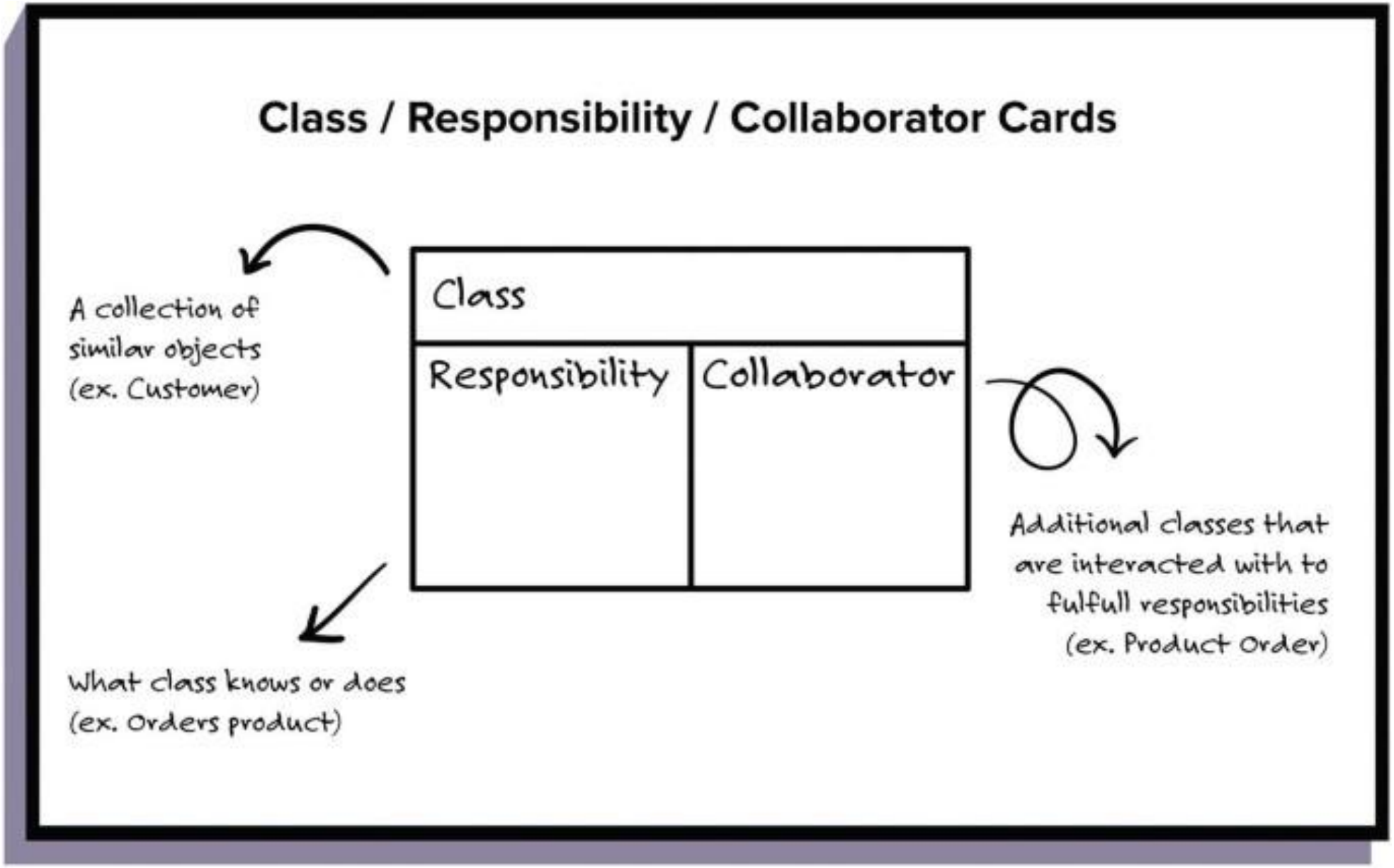
- Engineering of complex physical system is simplified by dividing the design into manageable smaller units.
- Similarly, engineering of software is also simplified by the identification and development of software components.
- A software component is an abstract entity that can perform some tasks, i.e., fulfill some responsibilities.
- A component may ultimately be turned into a function, a structure or class, or a collection of other components (module).
- Major characteristics:
    - must have a small well-defined set of responsibilities
    - should interact with other components to the minimal extent possible

Role of Behavior

- The design process in OOP begins with an analysis of behavior because the behavior of a system is usually understood at first hand, long before any other aspect.
- Conventional software development techniques focused on ideas such as characterizing the basic data structures or the overall structure of function calls, often within the creation of a formal specification of the desired application.
- Since structural elements of the software application can be identified only after a rigorous problem analysis, ironically, a formal specification often ended up as a document understood neither by the programmer nor the client (user).
- However, behavior is something that can be described almost from the moment an idea is conceived which can be described meaningfully to both programmers and clients.

CRC cards

- CRC stands for *Component/Class, Responsibility,* and *Collaborator*.
- Brainstorming tool used in the design of software using OOP.
- Small index cards used to represent components used when first determining which classes are needed and how they will interact.
- In order to discover software components and their responsibilities, the programming teams first walks through different scenarios and mocks the running of the software as if it is already developed – identifying every activity that must take place which is then assigned to some component as responsibility.
- Written on the face of a card is:
  - Name of the software component/class
  - Responsibilities of the component
  - Names of other components with which the component needs to interact


Class / Responsibility / Collaborator Cards
A collection of similar objects (ex. Customer)
Class
Responsibility
Collaborator
Additional classes that are interacted with to fulfull responsibilities (ex. Product Order)
What class knows or does (ex. Orders product)

An Example CRC Card

Class: *ShoppingCart*	
Responsibilities: Knows current items Knows the customer Add item Remove item	*Collaborators:* Customer ProductCatalog

Class: It describes the common properties of certain kinds of objects of interest in a particular problem. An object can be any abstract or real world entity. Each class must have a single, well-defined purpose that can be described clearly. The class-name is written across the top of the class with a short description of the purpose of the class written at the back of the card.

Responsibility: It is a service provided by an object of a class for other objects. It could either be something that must be done or something that must be known. To do something, an object makes use of its own knowledge and if that is insufficient, it takes help from other objects (its collaborators). The responsibilities of an object are written on the left of the card.

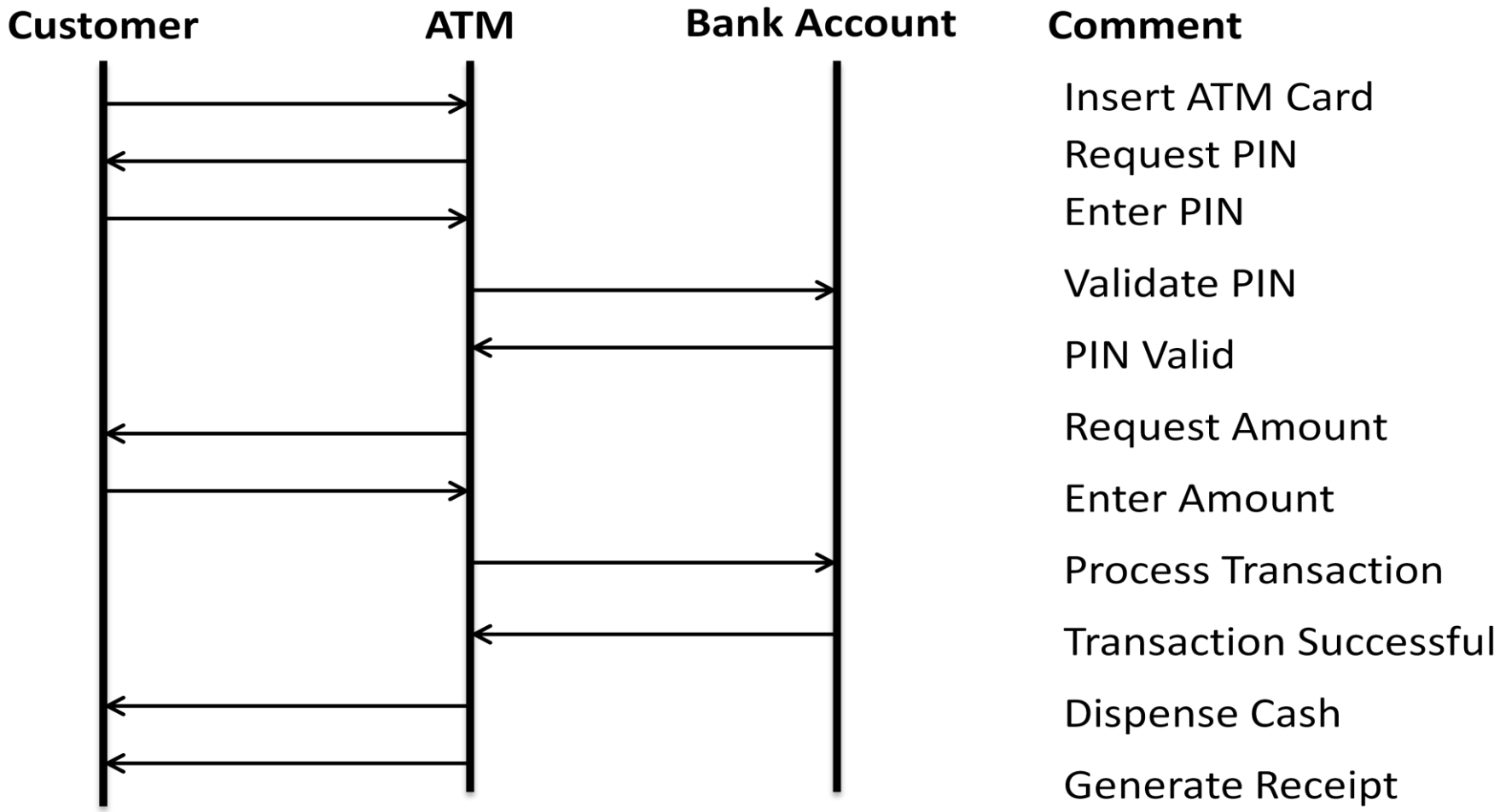
Collaborators: It indicates which objects can be asked for help to fulfill a specific responsibility. An object of the collaborator class can provide further information required for the completion of a particular responsibility or it can also take over the parts of the original responsibility.

Advantages

- **Simple and straightforward.**
- **Programming language independent.**
- **Easy to test** alternative analysis and design models long before the code is actually written.
- Provide a basis for more **formal analysis** and **design methodologies**.
- **Useful throughout the life cycle** of the software development process.
- **Portable**: Can be used anywhere, even away from the computer or office.
- **Increased user participation**.
- **Eases the transition** from process orientation to object orientation.
- Leads directly into creation of **class diagrams**.

Interaction Diagram / Sequence Diagram

- For describing the dynamic interaction during execution of a scenario
- Models the collaboration of components based on a time sequence
- In the diagram, time is shown to move forward from top to bottom
- Each component is represented by a labeled vertical line.
- A component sending a message to another component is represented by a horizontal arrow from one line (component) to another.
- A component returning back a value or control is represented by a reverse arrow.
- The comments on the right-most side explains about the interaction



A simplified example of Interaction Diagram
Scenario: Withdrawing Money from ATM

Different Aspects of Software Components

- Behavior and State
- Instance and Classes
- Coupling and Cohesion
- Interface and Implementation

i) Behavior and State

- Software components are characterized by their behavior: i.e., what they are capable of. Additionally, they may also hold certain information.
- **Behavior:** A set of actions a software component can perform. Complete description of all the behaviors for a component is also called the *protocol*.
- **State:** Represents all the information held within a component, which can change over time.
- Though it is not necessary for all components to maintain state information, most components consist of a combination of behavior and state.

ii) Classes and Instances

- **Class:** The term class is used to describe a set of objects with similar behavior. It is used as a syntactic mechanism in most object oriented programming languages to define a blueprint for objects belonging to the same class.
- **Instance:** An individual representative of a class is known as an instance or object.
- Behavior is associated with a class and not with an individual instance. All instances of a class will respond to the same instructions and perform in a similar manner. Behaviors are implemented as member functions in C++, which are also termed as methods.
- State is the property of an individual instance. All instances of a single class can perform the same actions but use different data values. State is implemented as data members in C++, which are also termed as properties.

iii) Cohesion and Coupling

- **Cohesion:** It is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating the tasks that are related in some manner into a single component.
- **Coupling:** It describes the degree of relationship / interconnectedness between software components. Coupling is increased when one software component must access the state (data values) held by another component.
- Since too much interconnection between software components obstruct ease of development, modification and reuse, it is desirable to reduce the amount of coupling and increase the degree of cohesion as much as possible.
- Low coupling or high cohesion can be achieved by moving a task into the responsibilities of the component which holds the necessary data.

iv) Interface and Implementation

- The most important achievement of characterizing a software component by its behavior is the advantage for one programmer to easily use a component developed by another programmer without needing to know how the component is implemented.
- The purposeful omission of implementation details behind a simple interface is known as *information hiding*.
- In other words, we can say that the component *encapsulates* the behavior, showing only how the component can be used, not the detailed actions it performs. This leads to two different views of a software system.
    - The *interface* view is the face seen by other programmers.
    - The *implementation* view is the face seen by the programmer working on the component.

- The separation of interface and implementation is one of the most important concepts in software engineering. This is mainly meaningful in programming in the large, where software components are often developed in parallel by different programmers, generally in isolation from each other. The limiting factor in such a scenario is the amount of communication required between various programmers and between their respective software components rather than the amount of coding involved.
- Furthermore, due to an increasing emphasis on the development of general-purpose software components for reuse across multiple projects, there should be minimal and well-understood interconnection between various portions of the system which is known as ***Parnas's principles*** **[David Parnas]**:

- The developer of a software component must provide the indented user with all the information needed to make effective use of the services provided by the component, and should be provided with no other information.
- The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

Formalizing the Interface

- Decision made regarding the general structure that will be used to implement each component (uniformity).
- A component with just one behavior and no internal state may be converted into a function whereas a component with many tasks can be implemented as a class.
- Names to be assigned to each of the responsibilities identified on the CRC card of each component, which are eventually mapped to function names
- The data required by a component to perform a specific task:
    - the source of data (global or argument or maintained internally)
- Identification of the types of arguments to be passed to the functions
- Description of information maintained within the component itself

Coming up with names

- Names associated with various activities should be carefully chosen such that they are useful in creating the vocabulary with which the eventual design will be formulated.
- Names should be internally consistent, meaningful, preferably short and suggestive in the context of the problem.
- Often, a considerable amount of time is spent finding just the right set of terms to describe the tasks performed and the object manipulated.
- Proper naming early in the design process greatly simplifies and facilitated the later steps of software development.

General Naming Guidelines [Keller 1990]:

- Use pronounceable names.
- Use capitalization or underscores to mark the beginning of a new word within a single name: e.g. PolarCoordinate or Polar_coordinate
- Be careful with abbreviations. An abbreviation obvious to one individual may be confusing to another.
- Avoid names with several interpretations. E.g., Empty( ): is it used to empty a data structure or check if it is empty or not.
- Avoid numeric digits within a name as far as practicable because some digits may be misread with alphabets (0 and o, 1 and l, 2 and Z, 5 and S).
- Name functions and variables that yield boolean values to reflect the interpretation as true or false. E.g., better to use IsEmpty( ) rather than Empty( ).
- Take extra care in the selection of names for operations that are costly and infrequently used.

Designing the Representation of Components

- It is an intermediate step towards transforming the description of a component into a software system implementation by dividing the design team into groups, each responsible for one or more software components.
- Major portion is designing the data structures used by each subsystem to maintain the state information required to fulfill the assigned responsibilities.
- The selection of appropriate data structure is an important task, central to the software design process, resulting into the code used by a component in the fulfillment of a responsibility self-evident.
    - Wrong choice of data structure = complex and inefficient programs
- Additional job is the transformation of descriptions of behavior into algorithms
    - Description to be matched against expectations of each component listed as collaborator, ensuring that expectations are fulfilled and necessary data items are available to carry out each process.

Implementing Components

- Once the design of each software subsystem is laid out, next step is to implement each component's desired behavior.
- Having the previous steps addressed correctly with each responsibility or behavior characterized with short description, major task is to implement the desired behavior using a computer language.
- If not determined earlier, decisions are to be made at this stage on issues that are entirely self-contained within a single component.
- Since it is rare that any one programmer will work on all aspects of a system, it becomes necessary i) to understand how one section of code fits into a larger framework and ii) to work well with other team members.
- Implementation of one component might require another component (facilitator) to work behind the scene. Thus requiring pre-condition to be documented well.

Integration of Components

- Once software subsystems have been individually designed and tested, they can be integrated into the final product.
- This is often not a single step, but a part of a larger process.
- Starting from a simple base, elements are slowly added to the system & tested.
- Often, use of stubs (simple dummy routines with no behavior or with very limited behavior) for yet unimplemented parts. Later on, replaced by more complete code.
- Testing of individual components = *Unit Testing*
- Further testing until the system is working as desired = *Integration Testing*
- Errors in a component might be manifested by errors in another component, thus necessitating fixing the implementation of buggy component.
- Re-execution of the test cases after such fixture or addition of other components = *Regression Testing*