Object Oriented Design

Chapter 7

Responsibility implies non-interference

- Responsibility implies a degree of independence or non-inference
- Example: when you send flowers to your grandfather at Pokhara, it was not necessary to think how the request would be served. The florist, having taken the responsibility for this service, is free to operate without inference from the customer

- In Conventional programming one portion of code is dependent on many other portions of the code of system
- Such dependencies arise through use of global variables, pointer variables or inappropriate use of and dependencies of implementation details on other portion of code
- Responsibility driven design make the portion of code as independent as possible
- Responsibility driven design support information hiding ans it becomes more important when one move from programming in small to programming in large

Programming in Small

- Code is developed by a single programmer or very small number of programmers

- A single individual can understand all the aspects of a project

Programming in Large

- Software system is developed by a large team
- Team consists of people with many different skills
- A single individual may not necessarily understand all aspects of the project
- Major problem in software development is team management and communication of information between different parts of the project

Role of behavior in OOP: why begin with behavior

- Object oriented design process begins with the analysis of behavior
- Behavior can be described from the moment an idea is conceived and can be described in terms meaning to both the programmers and the client

Responsibility Driven Design (RDD)

- Developed by Rebecca Wirfs- Brock
- It is an object oriented design technique that is driven by a emphasis on behavior at all levels of development
- In object oriented design, first we establish who is responsible to for which action of the system
- The responsible class to carry out an action may communicate with other classes to complete its responsibility
- Responsibility is anything that a class knows or does

• Example: class Student might have knowledge for its data members : registration number, roll number, name, address, faculty

and it may perform actions like attend class, take exam, view result, pay fee

Collaborator

- Collaborator is another class that is used to interact with a class

- Collaborator for a class is also a class which helps to perform first class’s action

Class Responsibility Collaborator (CRC)

- CRC card:

Identify class

Identify list of responsibility

List collaboration

Class Responsibility Collaborator Card

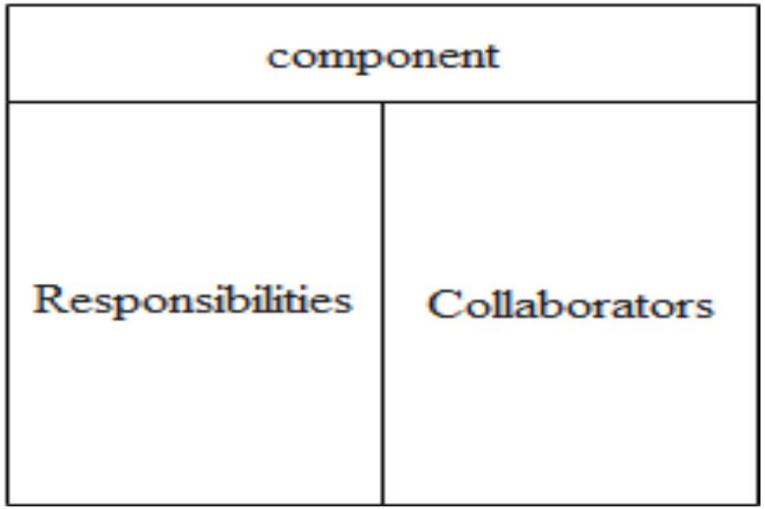
- Consists of three sections:

1. Class : a class represents a collection of similar objects

2. Responsibility: it is something that a class knows or does

3. Collaborator: another class that is a class interacts with to fulfill its responsibilities

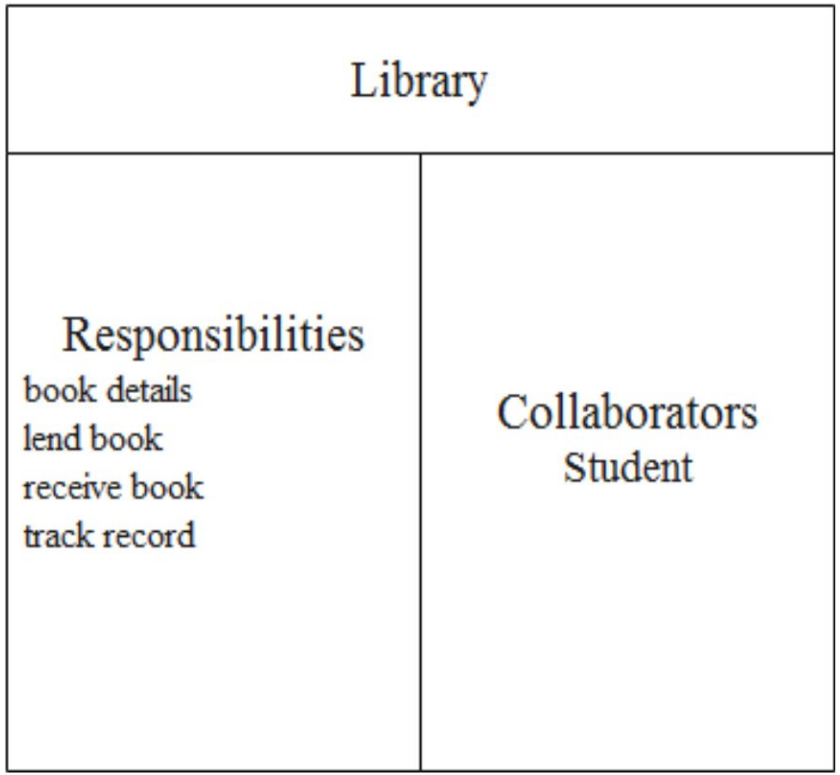
CRC card:



CRC card for Student:

Student	
Responsibilities name roll number registration number attend class fill examination form take exam request for transcript pay fee borrow book return book	**Collaborators** ExamDepartment AccountDepartment Library

CRC card for Library:



ATM CRC card example

Card reader	
Tell ATM when card is inserted Read information from card Eject card Retain card	**ATM Card**

Software Components

In programming and engineering disciplines, a component is an identifiable part of a larger program or construction.

Usually, a component provides a particular function or group of related functions.

In programming design, a system is divided into components that in turn are made up of modules.

Component test means testing all related modules that form a component as a group to make sure they work together.

In object-oriented programming , a component is a reusable program building block that can be combined with other components to form an application. Examples of a component include: a single button in a graphical user interface, a small interest calculator, an interface to a database manager.

) Behavior and State: One way to view a component is as a pair consisting of behaviour and state:

" The behavior of a component is the set of actions it can perform. The complete description of all the behavior for a component is sometimes called the protocol. For the Recipe component this includes activities such as editing the preparation instructions, displaying the recipe on a terminal screen, or printing a copy of the recipe.

" The state of a component represents all the information held within it at a given point of time. For our Recipe component the state includes the ingredients and preparation instructions. Notice that the state is not static and can change over time. For example, by editing a recipe (a behavior) the user can make changes to the preparation instructions (part of the state).

ii) Coupling and Cohesion

Two important concepts in the design of software components are coupling and cohesion.

Cohesion is the degree to which the responsibilities of a single component form a meaningful unit. High cohesion is achieved by associating in a single component tasks that are related in some manner. Probably the most frequent way in which tasks are related is through the necessity to access a common data value. This is the overriding theme that joins, for example, the various responsibilities of the Recipe component.

Coupling, on the other hand, describes the relationship between software components. In general, it is desirable to reduce the amount of coupling as much as possible, since connections between software components inhibit ease of development, modification, or reuse

iii) Interface and Implementation

It is very important to know the difference between interface and implementation. For example, when a driver drives the car, he uses the steering to turn the car. The purpose of the steering is known very well to the driver, but the driver need not to know the internal mechanisms of different joints and links of various components connected to the steering. An interface is the user's view of what can be done with an entity. It tells the user what can be performed.

Implementation takes care of the internal operations of an interface that need not be known to the user. The implementation concentrates on how an entity works internally.

Comparison of interface and implementation

Interface	Implementation
• It is user's viewpoint. (**What** part) • It is used to interact with the outside world. • User is permitted to access the interfaces only. • It encapsulates the knowledge about the object.	• It is developer's viewpoint. (**How** part) • It describes how the delegated responsibility is carried out. • It describes how the delegated responsibility is carried out. • It provides the restriction of access to data by the user.

Formalizing the interface

" The first step in this process is to formalize the patterns and channels of communication.

" A decision should be made as to the general structure that will be used to implement each component. A component with only one behavior and no internal state may be made into a function. Components with many tasks are probably more easily implemented as classes. Names are given to each of the responsibilities identified on the CRC card for each component, and these will eventually be mapped onto method names. Along with the names, the types of any arguments to be passed to the function are identified.

" Next, the information maintained within the component itself should be described. All information must be accounted for. If a component requires some data to perform a specific task, the source of the data, either through argument or global value, or maintained internally by the component, must be clearly identified.

Coming up with names

- Names should be internally consistent, meaningful, preferably short, and evocative in the context of the problem.

The following general guidelines have been suggested:

- Use pronounceable names. As a rule of thumb, if you cannot read a name out loud, it is not a good one.
- Use capitalization (or underscores) to mark the beginning of a new word within a name, such as “CardReader” or “Card reader”, rather than the less readable “cardreader”.
- Examine abbreviations carefully. An abbreviation that is clear to one person may be confusing to the next. Is a “TermProcess" a terminal process, something that terminates processes, or a process associated with a terminal?
- Avoid names with several interpretations.

Design and Representation of Components

- The task now is to transform the description of a component into a software system implementation major portion of this process is designing the data structures that will be used by each subsystem to maintain the state information required to fulfil the assigned responsibilities
- It is here that the classic data structures of computer science come into play. The selection of data structures is an important task, central to the software design process. Once they have been chosen, the code used by a component in the fulfilment of a responsibility is often almost self evident. But data structures must be carefully matched to the task at hand. A wrong choice can result in complex and inefficient programs, while an intelligent choice can result in just the opposite.

Design and Representation of Components

- It is also at this point that descriptions of behavior must be transformed into algorithms. These descriptions should then be matched against the expectations of each component listed as a collaborator, to ensure that expectations are fullled and necessary data items are available to carry out each process.

Implementation of Components

- The next step is to implement each component's desired behavior. If the previous steps were correctly addressed, each responsibility or behavior will be characterized by a short description. The task at this step is to implement the desired activities in a computer language.
- An important part of analysis and coding at this point is characterizing and documenting the necessary preconditions a software component requires to complete a task, and verifying that the software component will perform correctly when presented with legal input values.

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 part of a larger process. Starting from a simple base, elements are slowly added to the system and tested, using stubs: simple dummy routines with no behavior or with very limited behavior: for the as yet unimplemented parts.
- Testing of an individual component is often referred to as unit testing.
- Integration testing can be performed until it appears that the system is working as desired.
- Re-executing previously developed test cases following a change to a software component is sometimes referred to as regression testing.
- Give example of car making with different components bumper, gear, engine etc.

Sequence Diagram

- Sequence diagrams is used to model the interactions between objects in a single use case

- They illustrate how the different parts of a system interact with each other to carry out a function.

- Lifeline

A lifeline represents an individual participant in the Interaction

- Activations

A thin rectangle on a lifeline) represents the period during which an element is performing an operation

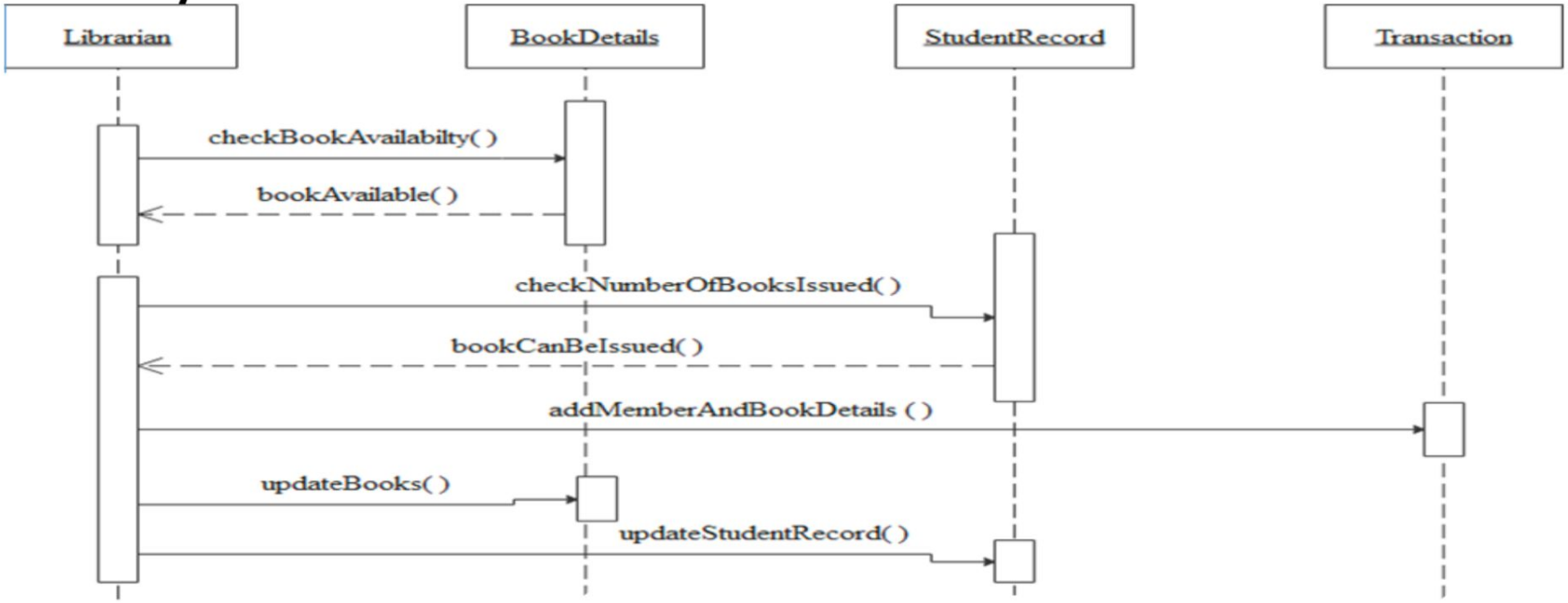
- CallMessage

A message defines a particular communication between Lifelines of an Interaction

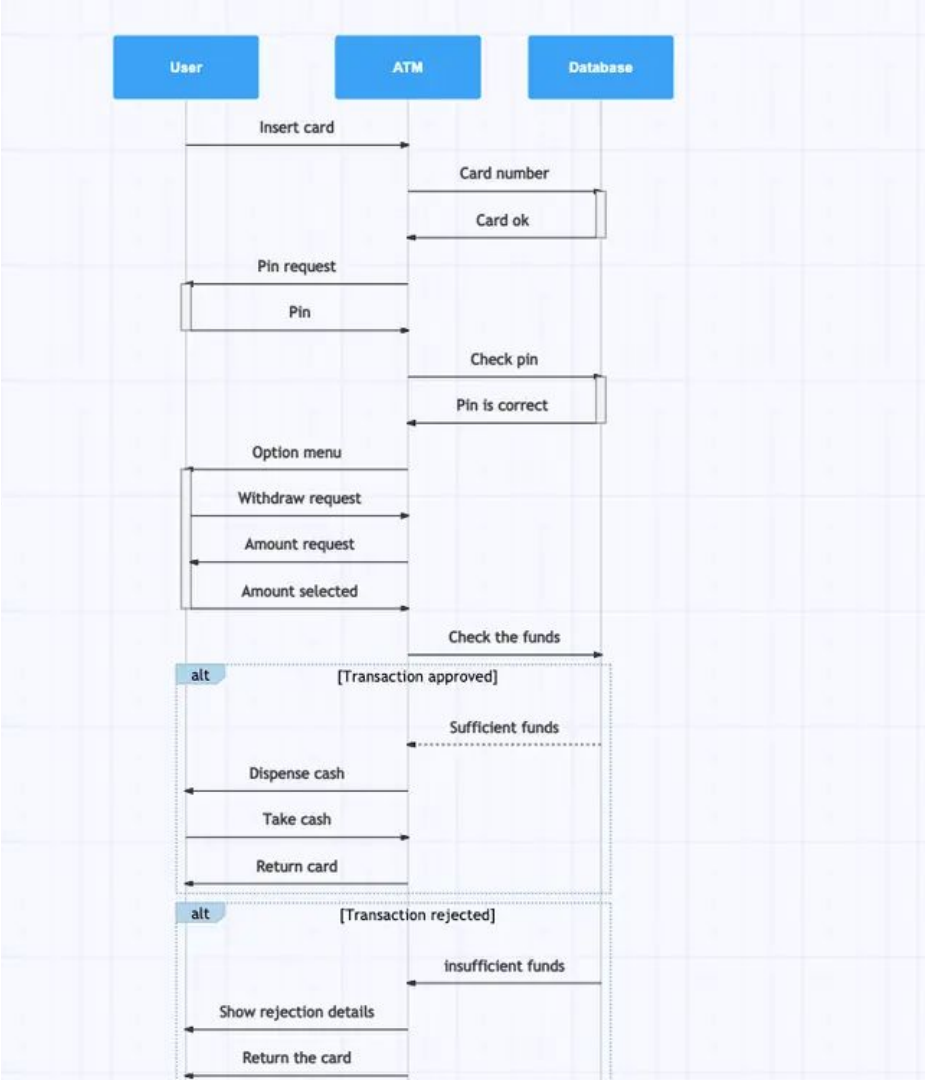
- ReturnMessage

Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message

Sequence diagram of borrowing book from library:



Draw the sequence diagram for withdrawing from ATM.



- The system starts when a customer presents a cheque to the clerk.
- The clerk checks the ledger containing all account numbers to validate the account number on the cheque and ensure it is valid.
- The clerk also checks the ledger for the account balance to verify if there is adequate balance to pay the cheque.
- If everything is valid, the clerk updates the ledger by debiting the customer's account by the specified amount on the cheque.
- If there is an error in the cheque, it is returned, and a Cheque Error Notification is generated.
- Once the cheque is verified, the clerk writes a token number on the top of the cheque and passes it to the cashier.
- The cashier calls out the token number to identify the customer and takes their signature.
- The cashier then pays cash to the customer.
- The cashier enters the cash paid information in the Day Book, which is a ledger for recording daily transactions.
- Finally, the cheque is filed for record-keeping purposes.