

Chapter 2 Requirement Analysis and Modeling

1.1 System Requirements

Hardware Requirements

Software Requirements

Hardware Requirements

1. Architecture Processing Power (CPU)
2. Memory (RAM)
3. Secondary Storage (Hard Disk)
4. Display adapter

Software requiring a better than average computer graphics display, like graphics editors and high-end games, often define high end display adapter.

5. Peripherals

Include CD-ROM drives, keyboards, pointing devices, network devices, etc.

Software Requirements

1. Platform
2. Web browsers
3. API's Drivers

Software making extensive use of special hardware devices, like high-end display adapters, needs special API or newer device drivers. A good example is DirectX, which is a collection of APIs for handling tasks related to multimedia, especially game programming, on Microsoft platforms

■ **Other Types of Requirements:**

- ❖ **Functional requirement**
- ❖ **Non Functional requirement**

- ❖ **Functional requirement:** Describes Functionality provided by the system.

Example: Banking System

Functional Requirements: User Bank account creation, withdrawal of money, Cheque Clearance

- ❖ **Non-Functional requirement:** Describes Constraints which needs to be considered while designing the system.

Example:

- Availability

Example: ATM System

Non-Functional requirement: System Must be available 24hrs. a day 7 days a week

- Reliability
- Security
- Maintainability
- Response Time
- Technology
- Platform
- Cost
- Delivery date

1.2 Stakeholders

+ Stakeholders

Any person or entity interested in a particular business is called a stakeholder. They are affected by the **business activity**, and they may be part of the core decision-making team. Internal and external stakeholders may have different interests and priorities, possibly leading to conflicts of interest.





Figure 2.2 Stakeholders

✚ Kinds of Stakeholders

1. **Internal stakeholders** are owners, managers and workers.
2. **External stakeholders** are the customers and the suppliers.
3. **The community** in which the organization does business also is a stakeholder.

Not all stakeholders are equal, and different stakeholders will have varying considerations.

These stakeholders can have direct or indirect stake in the organization and in policy-making.





Figure 2.2 Kinds of Stakeholders

✚ Influence of Stakeholders

1. Different stakeholders have different influence.
2. Owners have a major say in the way the company functions.
3. **Owners** generally tend to extract the maximum efficiency and make the maximum profit from their investments in the company.
4. **Customers** are also key stakeholders in any organization. The way they are catered to and their level of satisfaction determines how the company runs.

✚ Primary and Secondary Stakeholders

1. Primary stakeholders are the most important people to the business.
2. In small businesses, primary stakeholders are **owners, staff and customers**.
3. These primary stakeholders decide the company policies and plans.
4. In large businesses, primary stakeholders can vote the directors out if they feel the directors are not performing properly.
5. **Less influential stakeholders** are referred to as secondary stakeholders.

✚ Stakeholder Interests

Various stakeholders have various interests in the company.

1. Owners want to maximize their profits and are interested in how well their business is functioning.

2. Managers and workers are interested in their salaries and want to keep their jobs at all costs.
3. Lenders want the businesses to repay their loans on time and in full.
4. Customers want the company to produce high-quality products for affordable rates. They also look for good customer service before and after the sale.
5. The community wants the company to be environmentally friendly.

1.3 Requirement gathering techniques

One-on-one interviews

- The most common technique for gathering requirements is to sit down with the clients and ask them what they need.
- The discussion should be planned out ahead of time based on the type of requirements you're looking for.
- There are many good ways to plan the interview, but generally you want to ask open-ended questions to get the interviewee to start talking and then ask probing questions to uncover requirements.

Group interviews

- Group interviews are similar to the one-on-one interview, except that more than one person is being interviewed -- usually two to four.
- These interviews work well when everyone is at the same level or has the same role.
- Group interviews require more preparation and more formality to get the information you want from all the participants.
- You can uncover a richer set of requirements in a shorter period of time if you can keep the group focused.

Facilitated sessions

- In a facilitated session, you bring a larger group (five or more) together for a common purpose.
- In this case, you are trying to gather a set of common requirements from the group in a faster manner than if you were to interview each of them separately.

Joint application development (JAD)

- JAD sessions are similar to general facilitated sessions.
- However, the group typically stays in the session until the session objectives are completed.

- For a requirements JAD session, the participants stay in session until a complete set of requirements is documented and agreed to.

Questionnaires

- Questionnaires are much more informal, and they are good tools to gather requirements from stakeholders in remote locations or those who will have only minor input into the overall requirements.
- Questionnaires can also be used when you have to gather input from dozens, hundreds, or thousands of people.

Prototyping

- Prototyping is a relatively modern technique for gathering requirements.
- In this approach, you gather preliminary requirements that you use to build an initial version of the solution -- a prototype.
- You show this to the client, who then gives you additional requirements.
- You change the application and cycle around with the client again.
- This repetitive process continues until the product meets the critical mass of business needs or for an agreed number of iterations.

Use cases

- Use cases are basically stories that describe how discrete processes work.
- The stories include people (actors) and describe how the solution works from a user perspective.
- Use cases may be easier for the users to articulate, although the use cases may need to be distilled later into the more specific detailed requirements.

Following people around

- This technique is especially helpful when gathering information on current processes.
- You may find, for instance, that some people have their work routine down to such a habit that they have a hard time explaining what they do or why.
- You may need to watch them perform their job before you can understand the entire picture. In some cases, you might also want to participate in the actual work process to get a hands-on feel for how the business function works today.

Request for proposals (RFPs)





- If you are a vendor, you may receive requirements through an RFP.
- This list of requirements is there for you to compare against your own capabilities to determine how close a match you are to the client's needs.


Brainstorming


- On some projects, the requirements are not "uncovered" as much as they are "discovered."
- Brainstorming sessions generate a lot of material that must be filtered and organized


❖ Brainstorming works by focusing on a problem, and then deliberately coming up with as many solutions as possible and by pushing the ideas as far as possible


❖ **There are four basic rules in brainstorming**

-  **No criticism**
-  **Welcome unusual ideas**
-  **Quantity Wanted**
-  **Combine and improve ideas**

 **No criticism:** The purpose is on generating varied and unusual ideas and extending or adding to these ideas. Criticism is reserved for the evaluation stage of the process. This allows the members to feel comfortable with the idea of generating unusual ideas.

 **Welcome unusual ideas:** Unusual ideas are welcomed as it is normally easier to "tame down" than to "tame up" as new ways of thinking and looking at the world may provide better solutions.

 **Quantity Wanted:** The greater the number of ideas generated, the greater the chance of producing a radical and effective solution.

 **Combine and improve ideas:** Not only are a variety of ideals wanted, but also ways to combine ideas in order to make them better.

Steps:

- ❖ **Gather the participants from as wide a range of disciplines with as broad a range of experience** as possible. This brings many more creative ideas to the session.
- ❖ Write down a brief description of the problem - the leader should take control of the session, initially defining the problem to be solved with any criteria that must be met, and then keeping the session on course.
- ❖ Use the description to get everyone's mind clear of what the problem is and post it where it can be seen. This helps in keeping the group focused

- ❖ Do NOT evaluate ideas until the session moves to the evaluation phase. Once the brainstorming session has been completed, the results of the session can be analyzed and the best solutions can be explored either using further brainstorming or more conventional solutions.
- ❖ The leader should keep the brainstorming on subject, and should try to steer it towards the development of some practical solutions.
- ❖ Once all the solutions have been written down, evaluate the list to determine the best action to correct the problem.

1.4 Requirement validation techniques

Validation Vs Verification:

Validation:

- The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with *verification*."
- **Validation** denotes **checking whether inputs** (context information), **performed activities** and **created outputs** (requirements artifacts) of the requirements engineering process fulfill predefined quality criteria.
- Validation is performed by involving **relevant stakeholders, other requirement sources** (e.g. standards, laws, ...) as well as **external reviewers**, if necessary.
- **Error propagation**
 - The earlier a requirement defect is detected, the cheaper it is to correct it...

Verification:

- The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with *validation*."

Quality criteria for requirements

1. **Complete**: no missing information
2. **Traceable**: source, evolution, impact and use
3. **Correct**: confirmed by stakeholders
4. **Unambiguous**: single valid interpretation
5. **Comprehensible**: content is easy to comprehend
6. **Consistent**: statements do not contradict each other
7. **Verifiable**: implemented system can be checked

8. **Rated:** relevance and/or stability have been determined and documented
9. **Up to date:** reflects the current status of the system in the system context

Validation vs. verification

■ **Boehm's definition:**

1. " **Validation:** Am I building the right system?
2. " **Verification:** Am I building the system right?

■ **Pohl's definition:**

1. " **Requirements verification:** Checking (by formal proofs) the correctness of statements about a requirements model of the system
2. " **Requirements validation:** Checking whether the right requirements have been defined (prior to requirements verification)

• **Validation in the content dimension:**

- Completeness (requirement)
- Consistency i.e. no contradictions
- Correctness (requirement) w.r.t. stakeholder desires
- Traceability

Validation in the documentation dimension :

Checking for adherence to the documentation rules and guidelines defined for the project:

- Correct documentation format
- Comprehensible documented requirements
- Unambiguous documentation
- Compliance with documentation rules

validation techniques:

1. **Inspections**
2. **Desk-checks**
3. **Walkthroughs**
4. **Prototypes**
 - ☐ **automatically generated**
 - ☐ **manually developed**

1: Inspections

- An organized examination process of the requirements

■ Involved roles:

- " Organizer (plan, monitor)
- " Moderator (neutral, i.e. not involved in artifact creation)
- " Author (explain)
- " Reader/presenter (independent)
- " Inspectors (for 4 context facets)
- " Minute-taker (document)

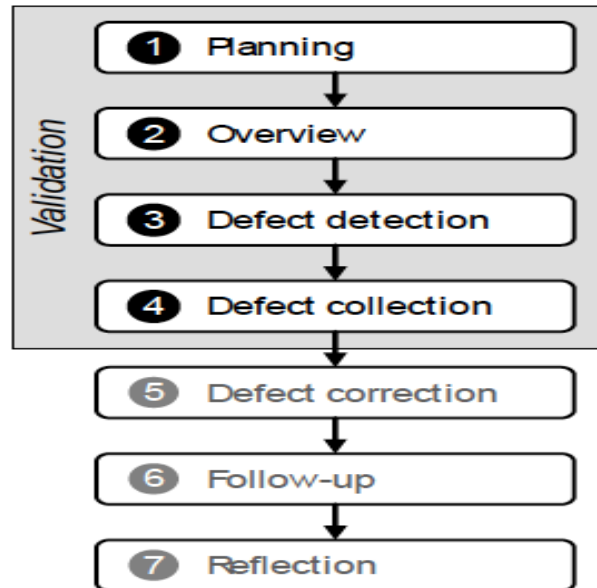


Figure 2.3 Inspection

2: Desk-checks:

1. The author of a requirement artifact distributes the artifact to a set of stakeholders
2. The stakeholders check the artifact individually
3. The stakeholders report the identified defects to the author
4. The collected issues are discussed in a group session (optional)

Same roles involved as for inspections

3: Walkthroughs:

A walkthrough does not have a formally defined procedure and does not require Differentiated role assignment.

- Checking early whether or not an idea is feasible
- Obtaining the opinion and suggestions of other people
- Checking the approval of others and reaching an agreement

Artifacts to be checked are not distributed beforehand

4: Prototypes:

- A prototype allows the stakeholders to try out the requirements for the system and Experience them thereby;
 1. Develop the prototype (tool support)
 2. Training of the stakeholders (experience)
 3. Observation of prototype usage (no influencing)
 4. Collect and analyze defects, identify conflicts
 5. If necessary, adapt prototype and validate again

Modeling System Requirements

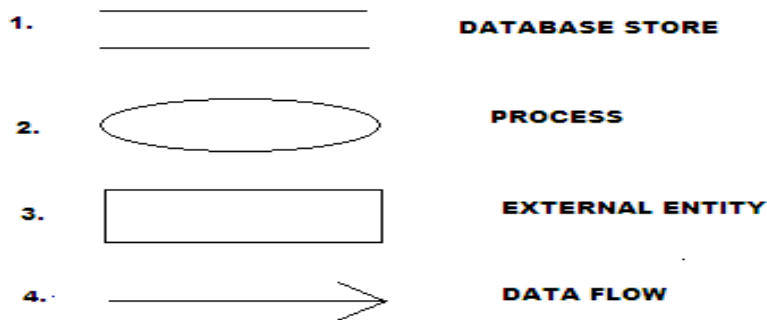
Traditional Approach to Requirement Modeling:

1. DATA FLOW DIAGRAM (DFD):

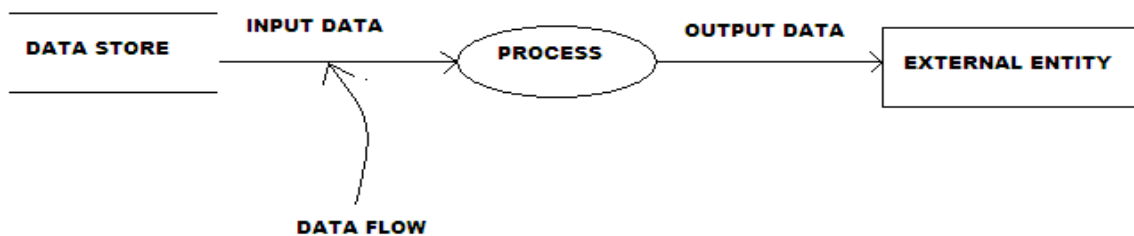
- Data flow diagram is a graphical model of system, which shows **what are the various functions (activities) performed by the system** and **how data flows among various functions**.
- **Each function is considered as a separate process.**
- A **data flow diagram (DFD)** is a graphical representation of the "flow" of data through an **information system**. DFDs can also be used for the **visualization** of **data processing** (structured design).
- On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, via an internal process.
- A DFD provides no information about the timing of processes, or about whether processes will operate in sequence or in parallel.
- It is therefore quite different from a **flowchart**, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed, in what order, and under what circumstances, but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

- Data Flow diagrams (DFD) that help you model data flows and functional requirements for a designed system.

- **Elements of Data Flow Diagram**



- 1. **DATA STORE**:-A data store is a holding place for information within the system
 2. **PROCESS**:-A process performs transformation or manipulation on input data and produces output data.
 3. **ENTITY**:-An entity is anything that interact with system and is a source or destination of a data.
 4. **DATA FLOW**:-A data flow shows the flow of information from its source to its destination.



Data flow diagram Example

■ DFD-LEVELS

■ **Level-0-DFD(Context Level DFD) :-**

- It is common practice to draw a [context-level data flow diagram](#) first, which shows the interaction between the system and external agents which act as data sources and data sinks.

- On the context diagram (also known as the 'Level 0 DFD') the system's interactions with the outside world are modeled purely in terms of data flows across the *system boundary*. **The context diagram shows the entire system as a single process, and gives no clues as to its internal organization.**

■ Level-1-DFD :-

- A Level 1 DFD that shows some of the detail of the system being modelled.
- The Level 1 DFD shows how the system is divided into sub-systems (processes), each of which deals with one or more of the data flows to or from an external agent, and which together provide all of the functionality of the system as a whole.
- It also identifies internal data stores that must be present in order for the system to do its job, and shows the flow of data between the various parts of the system.

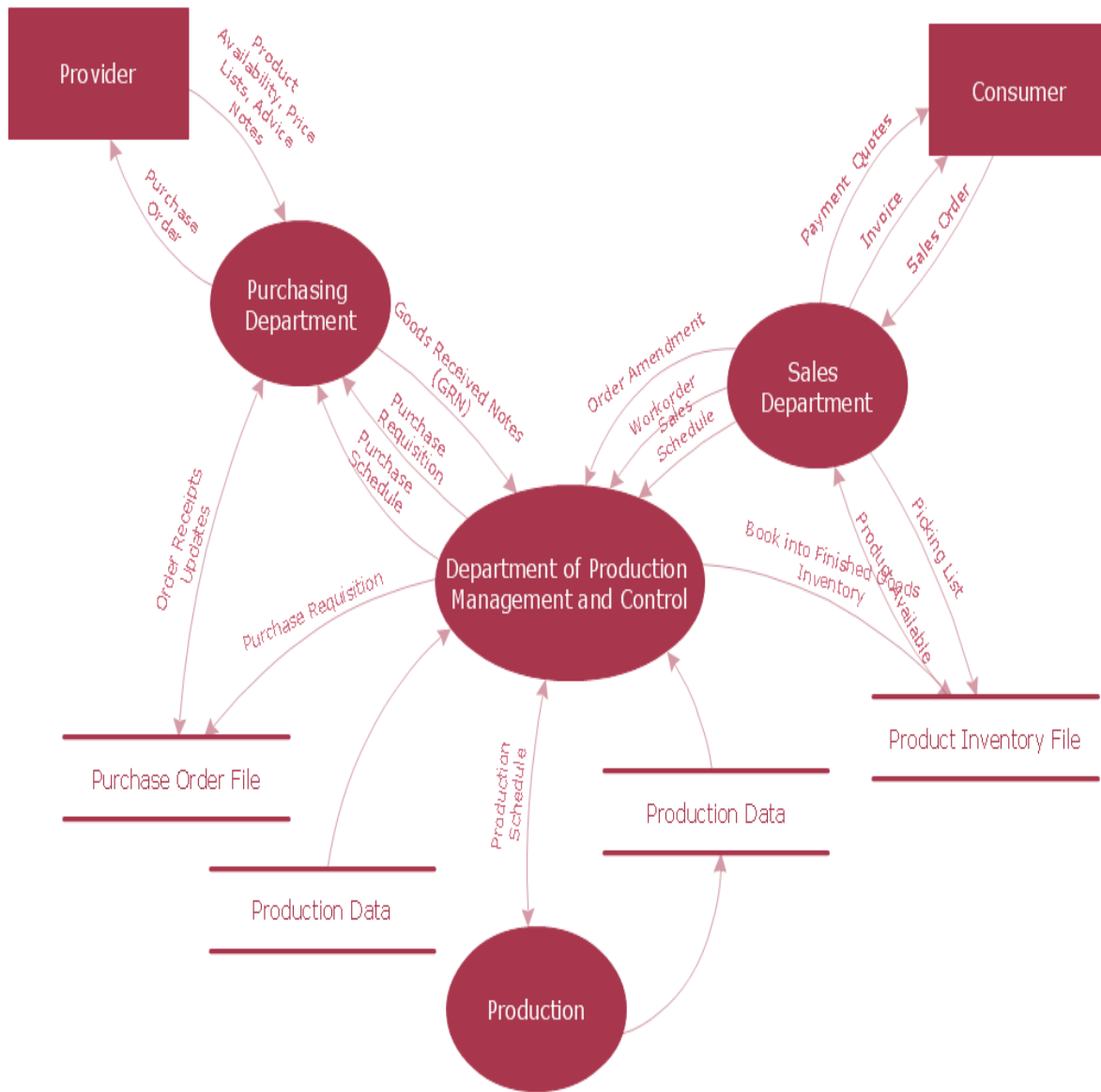
■ Level-2-DFD:-

- This level is a decomposition of a process shown in a level-1 diagram. There should be a level-2 diagram for each and every process shown in a level-1 diagram. i.e. it is breakdown of level-1-DFD.

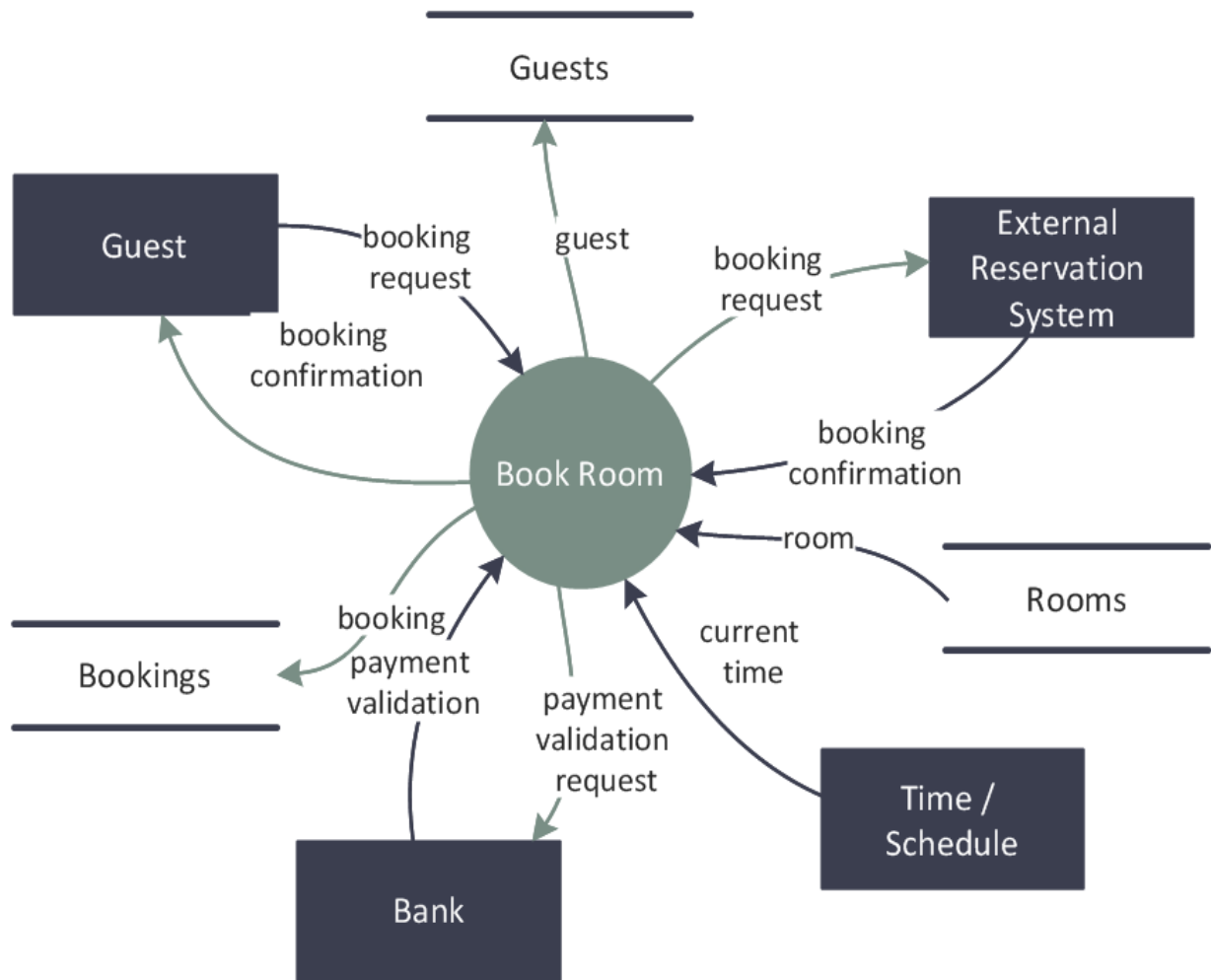
■ Level-3-DFD:-

- Level 3 DFD is a breakdown of Level 2 DFD.

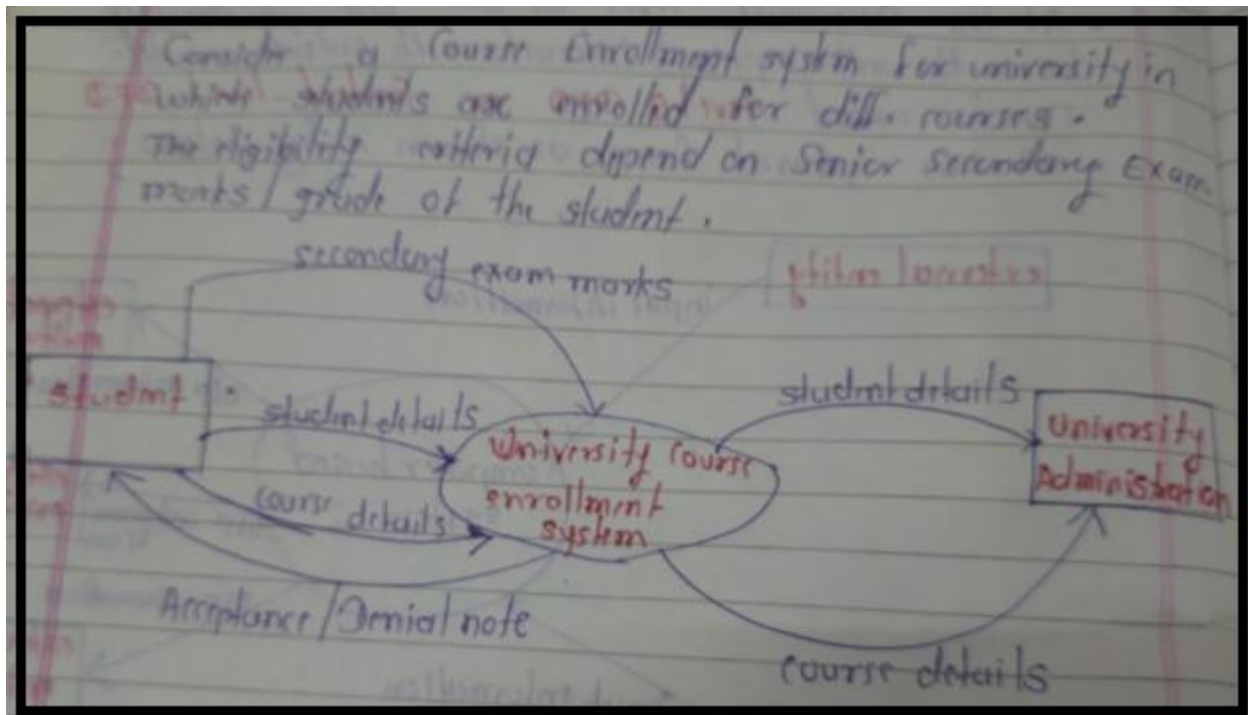
Example: This example shows the model of small traditional production enterprise. DFD diagrams are a useful way to visualize the system and what it will accomplish.(Level 1 DFD for Production Enterprise)



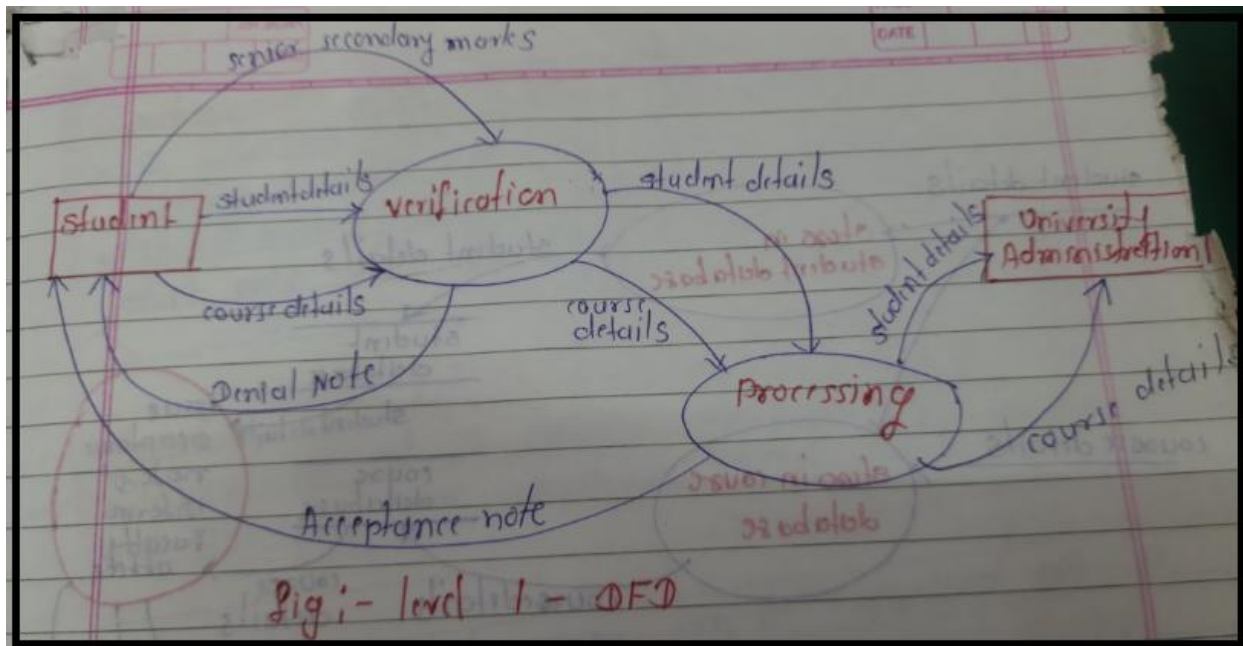
Example: Level 1 DFD for Hotel Management System



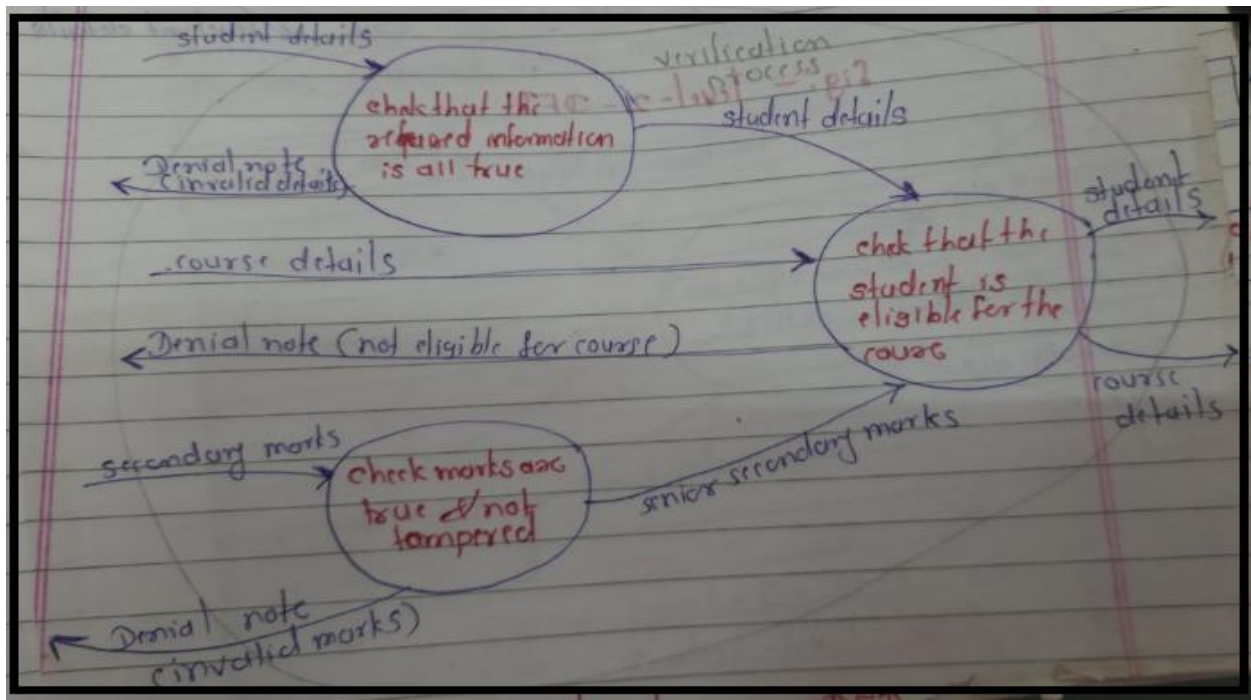
Example:



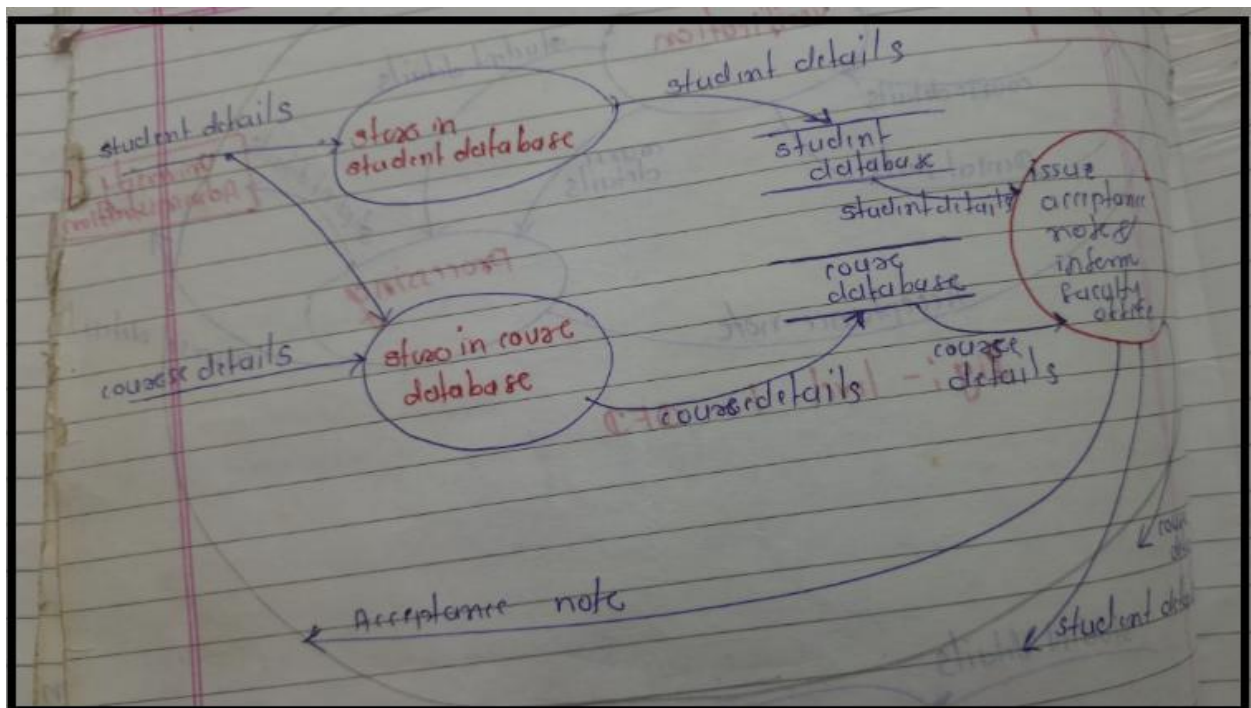
Level 0 DFD or Context level DFD



Level 1 DFD

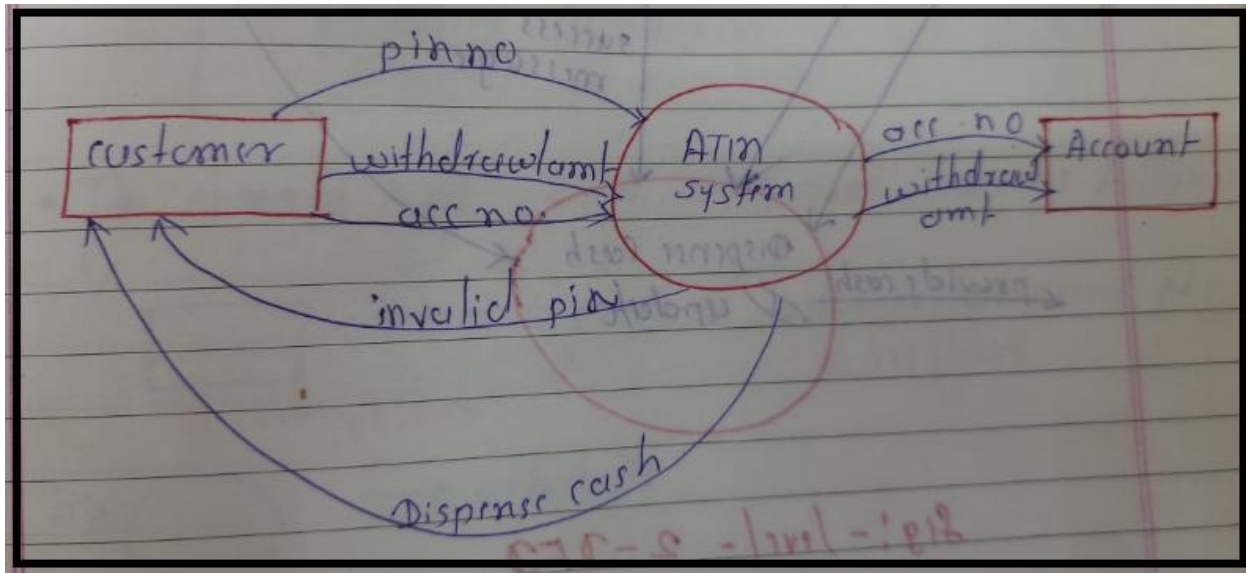
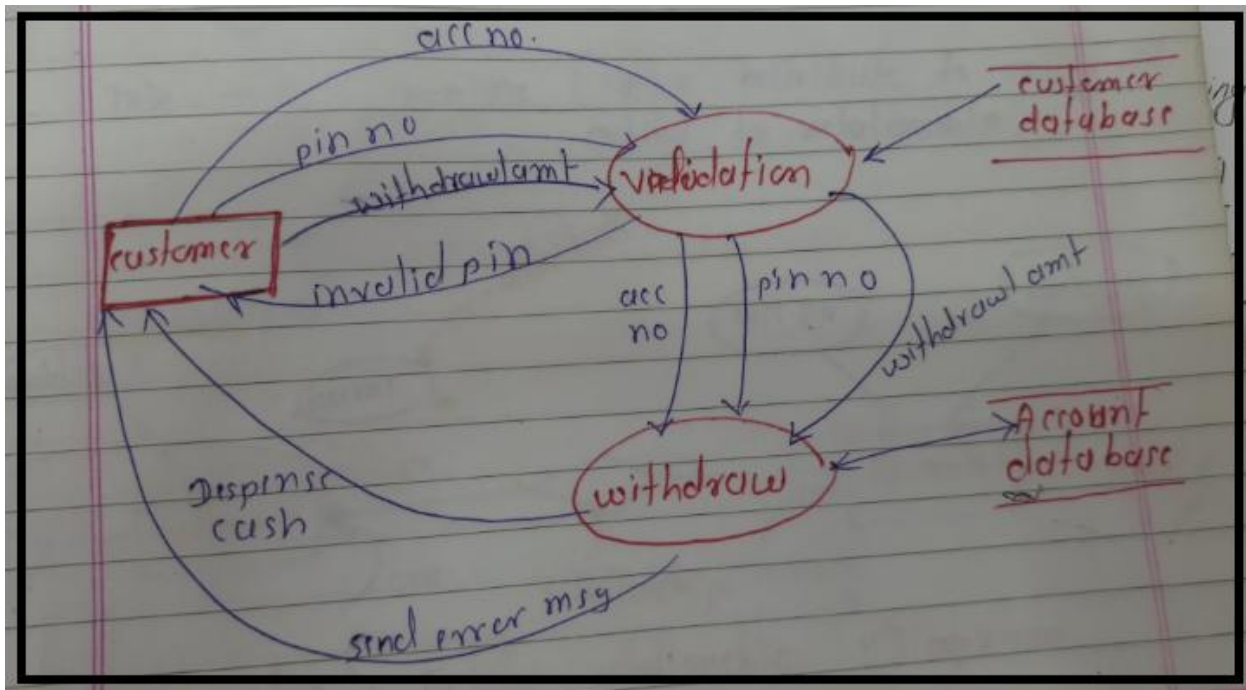


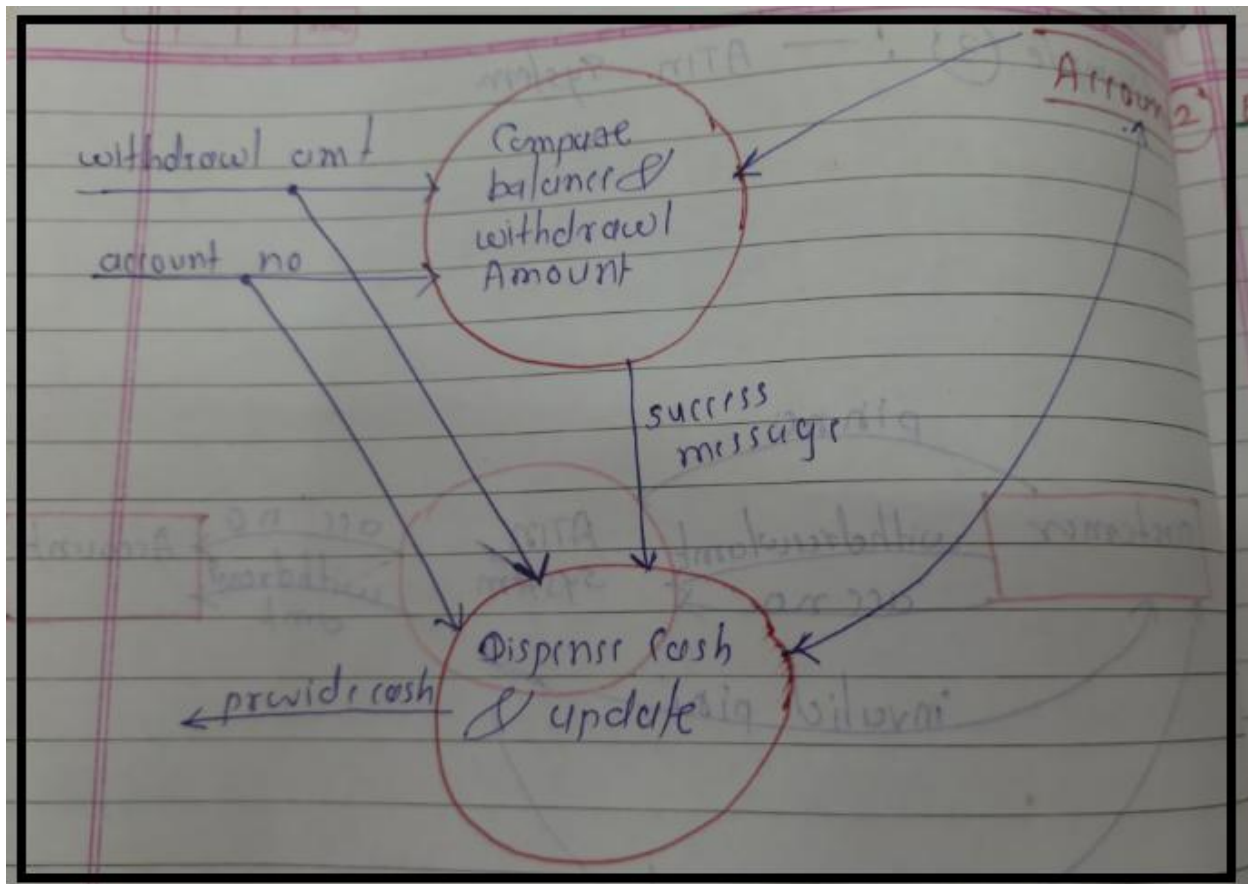
Level 2 DFD for verification Process of level 1 DFD



Level 2 DFD for Processing Process of level 1 DFD

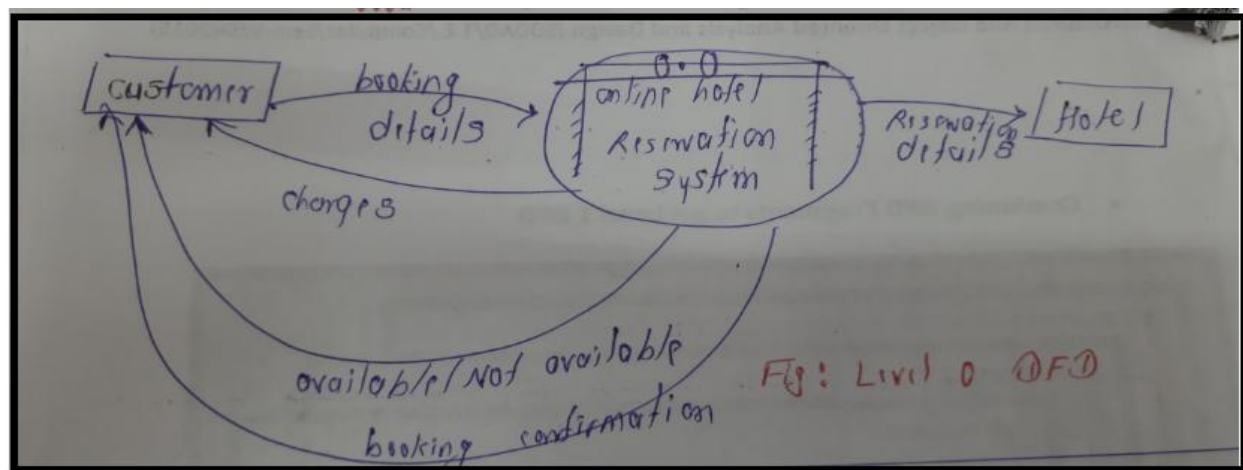
Example: ATM system

**Level 0 DFD****Level 1 DFD**

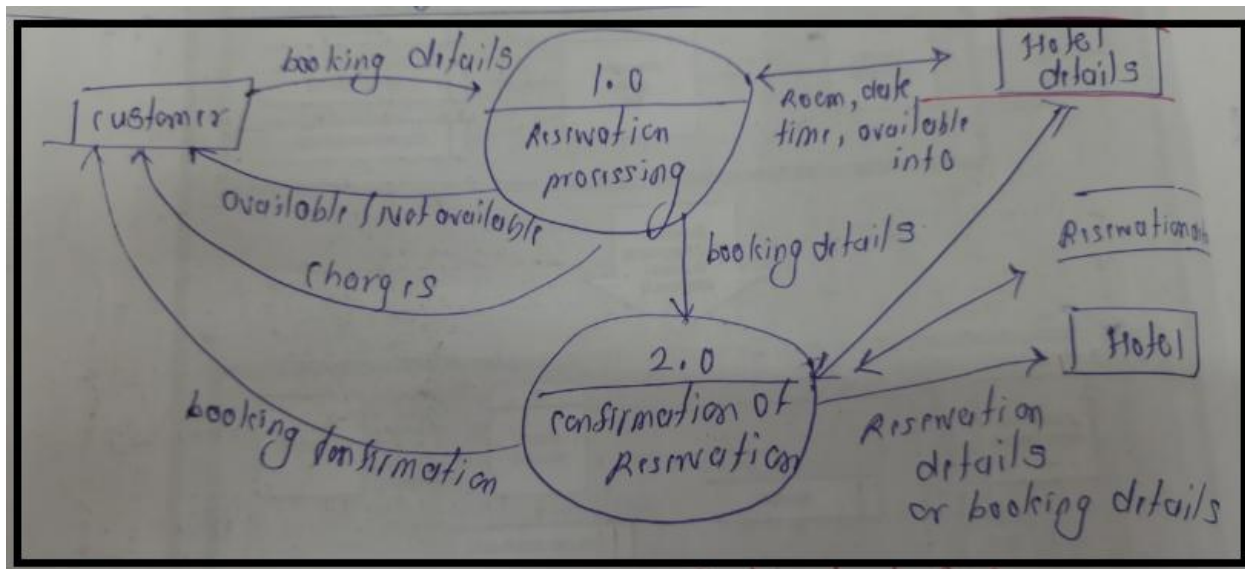
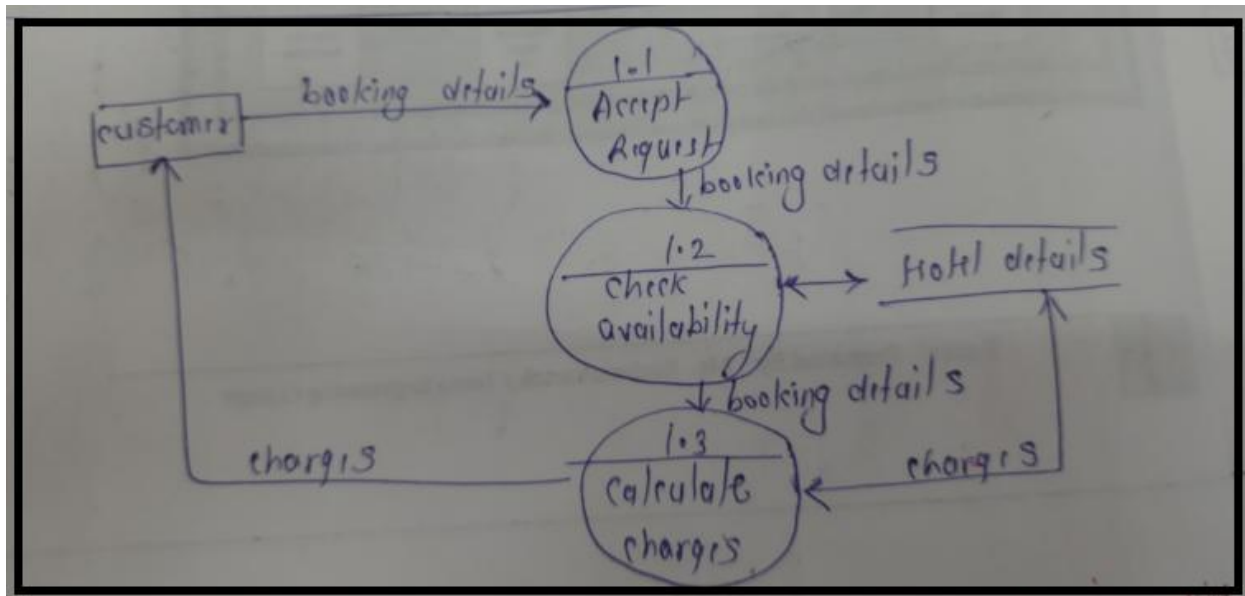


Level 2 DFD for withdrawal process of level 1 DFD

Example: Hotel Reservation System



Level 0 DFD

**Level 1 DFD****Level 2 DFD**

Object Oriented Approach to Requirements

■ UML(Unified Modeling Language)

- The Unified Modeling Language is a standard for **specifying, visualizing, constructing, and documenting** the artifacts of software systems, as well as for **business modeling and other non-software system**.
- 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 project teams communicate, explore potential designs, and validate the architectural design of the software.
- UML is modeling language used to prepare models of software system in the form of various UML diagrams.

■ Building Blocks of UML:

✓ **Things**

✓ **Relationships**

✓ **Diagrams**

✓ **Things:**

- Use Case
- Class
- Object
- State
- Activity
- Component
- Node
- Package
- Note
- Message

✓ **Relationships:**

- Dependency
- Generalization
- Association
- Aggregation
- Composition

✓ **UML diagrams:**

- Use Case Diagram
- Class Diagram
- Sequence Diagram
- Collaboration Diagram
- Activity diagram
- State Chart Diagram
- Component Diagram
- Deployment Diagram

■ **UML diagram classification:**

1. Static Diagram

- Use case diagram, Class diagram

2. Dynamic Diagram

- State diagram, Activity diagram, Sequence diagram, Collaboration diagram

3. Implementation Diagram

- Component diagram, Deployment diagram

■ **Use Case Diagram**

- Use Case diagram that shows **the interaction between user and system to capture the user's goals.**

- Use Case Diagram 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; summary of scenarios**
- 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

- UML Use case diagram consists of;

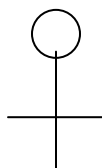
- + Use cases

- + Actors

- + Relationships(Interaction between the use case and actors).

- + Actors

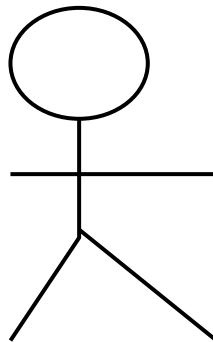
- Actors represent anyone or anything that interact with the system.
 - An actor may
 - Only input information to a system
 - Only retrieve information from a system
 - Both input and retrieve information to and from a system
 - Typically, the actors are found in the problem statement, and also from conversation with the customers and domain experts.
 - There are three types of actors:
 1. users of the system,
 2. external application systems, and
 3. External devices that can independently interact with the system.
 - In UML, an actor is represented **stickman symbol**, as shown below:



Actor

- **Examples:**

- Passenger: A person in the train
- GPS satellite: Provides the system with GPS coordinates



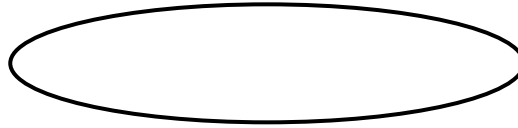
- **Use cases:**

- Use cases eventually map to the menu option. Use cases represent the functionality provided by the system. Each individual functionality provided by a system is captured as a use case.



- A use case represents a class of functionality provided by the system as an event flow.
- A use case consists of:
 - Unique name
 - Participating actors
 - Entry conditions
 - Flow of events
 - Exit conditions
 - Special requirements
 - **Examples:**

- Purchase Ticket



+ Relationships:

- Association

- Generalization

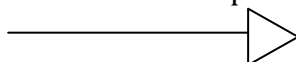
- Dependency

- The <<include>>Relationship (stereotype)
- The <<extend>>Relationship

- **Association:** communication between an actor and a use case; Represented by a solid line.



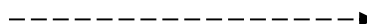
- **Generalization:** relationship between one general use case and a special use case (used for defining special alternatives). Represented by a line with a triangular arrow head toward the parent use case.

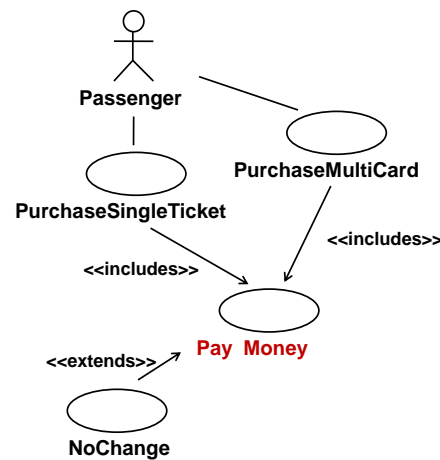


- **The <<include>>Relationship**

- A dotted line labeled<<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use “include” instead of copying the description of that behavior
- The direction of a <<includes>> relationship is to the using use case (unlike <<extends>> relationships).
- **Source use case explicitly uses behavior of another use case.**

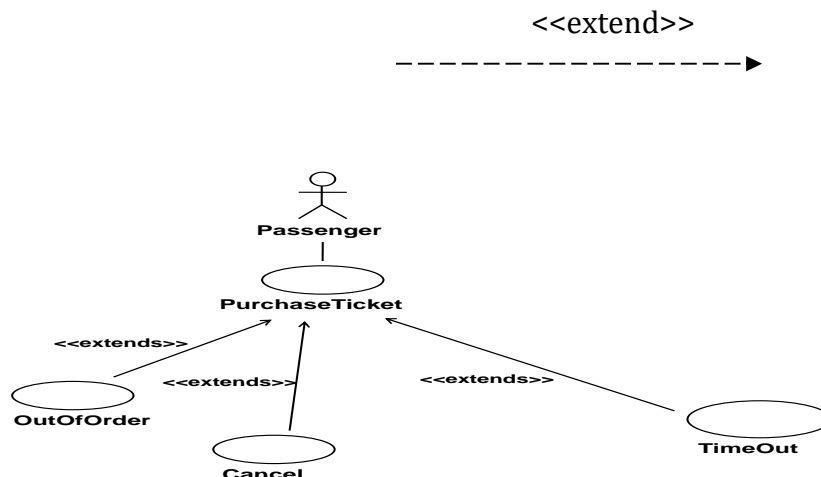
<<include>>

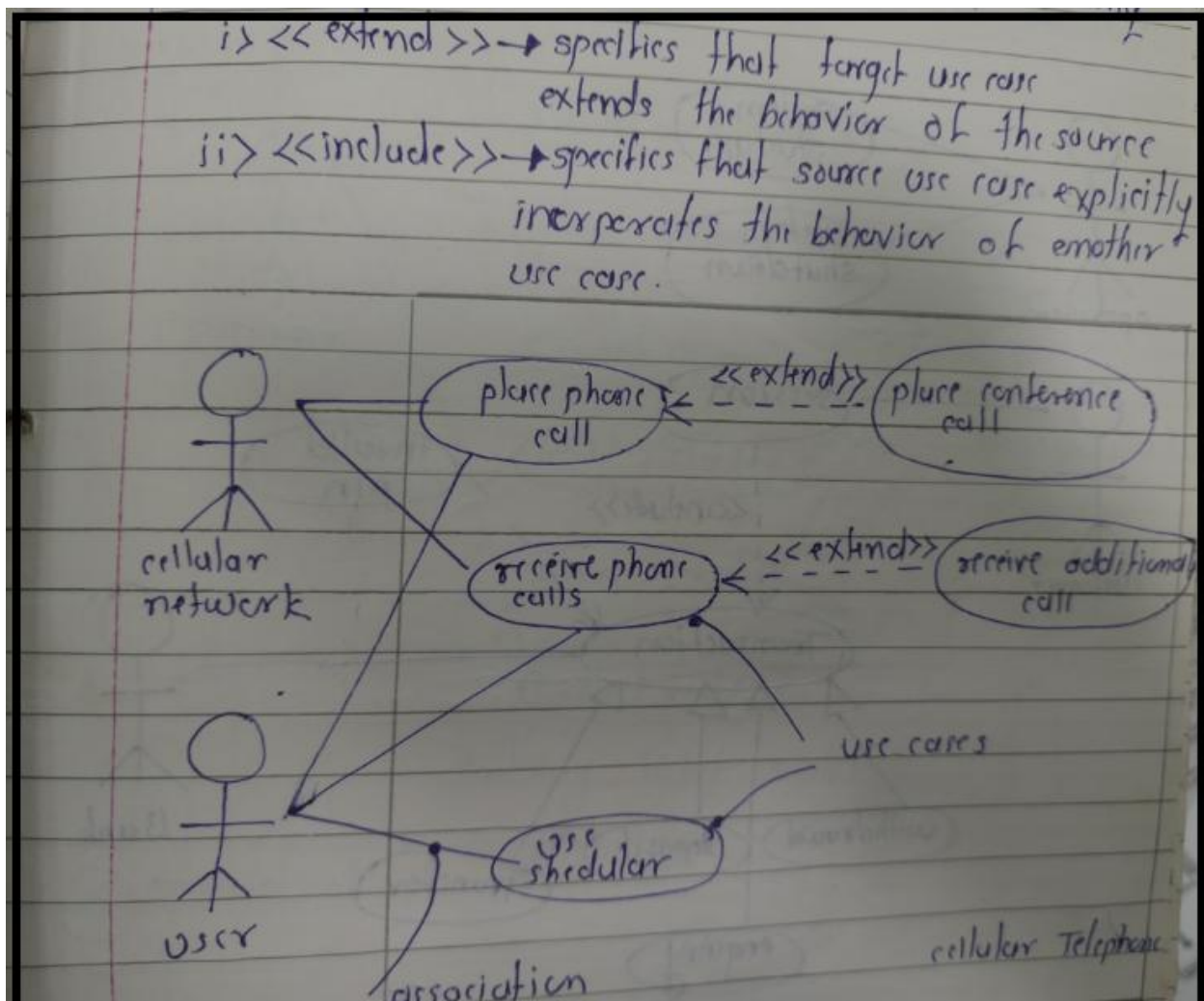




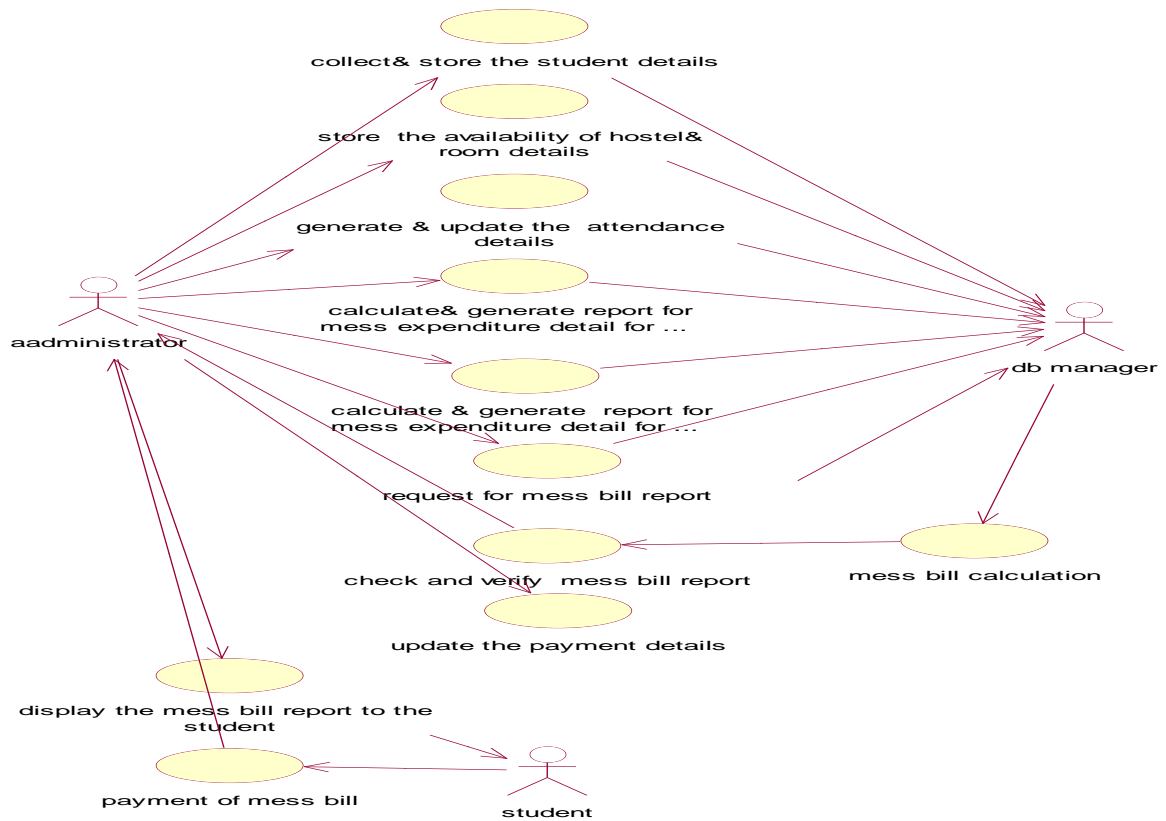
■ The <<extend>>Relationship

- A dotted line labelled <<extend>> with an arrow toward the base case. The extending use case may add behaviour to the base use case. The base class declares “extension points”.
- Use cases representing exceptional flows can extend more than one use case.
- The direction of a <<extends>> relationship is to the extended use case.
- **Target use case extends the behavior of source use case.**

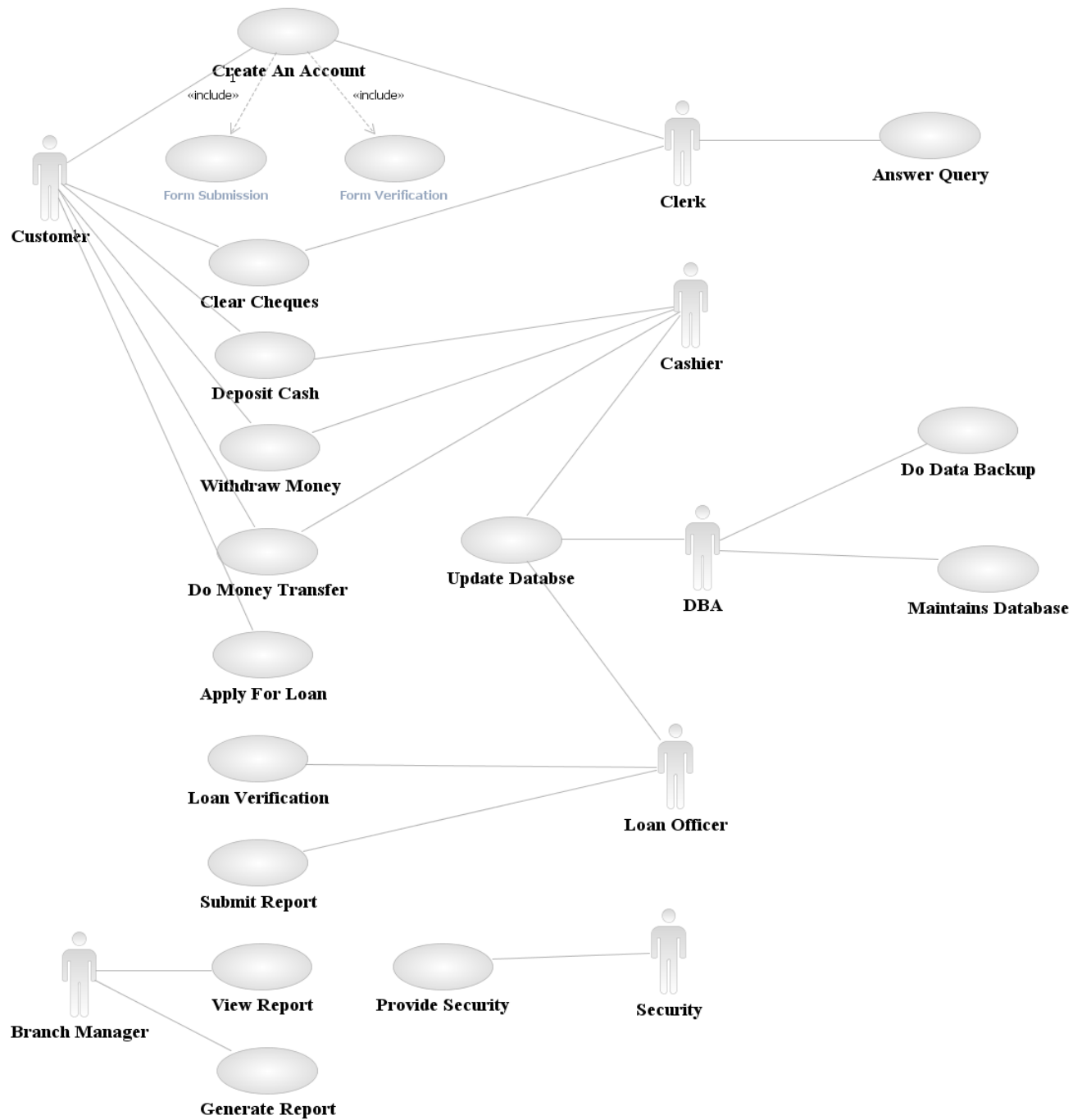




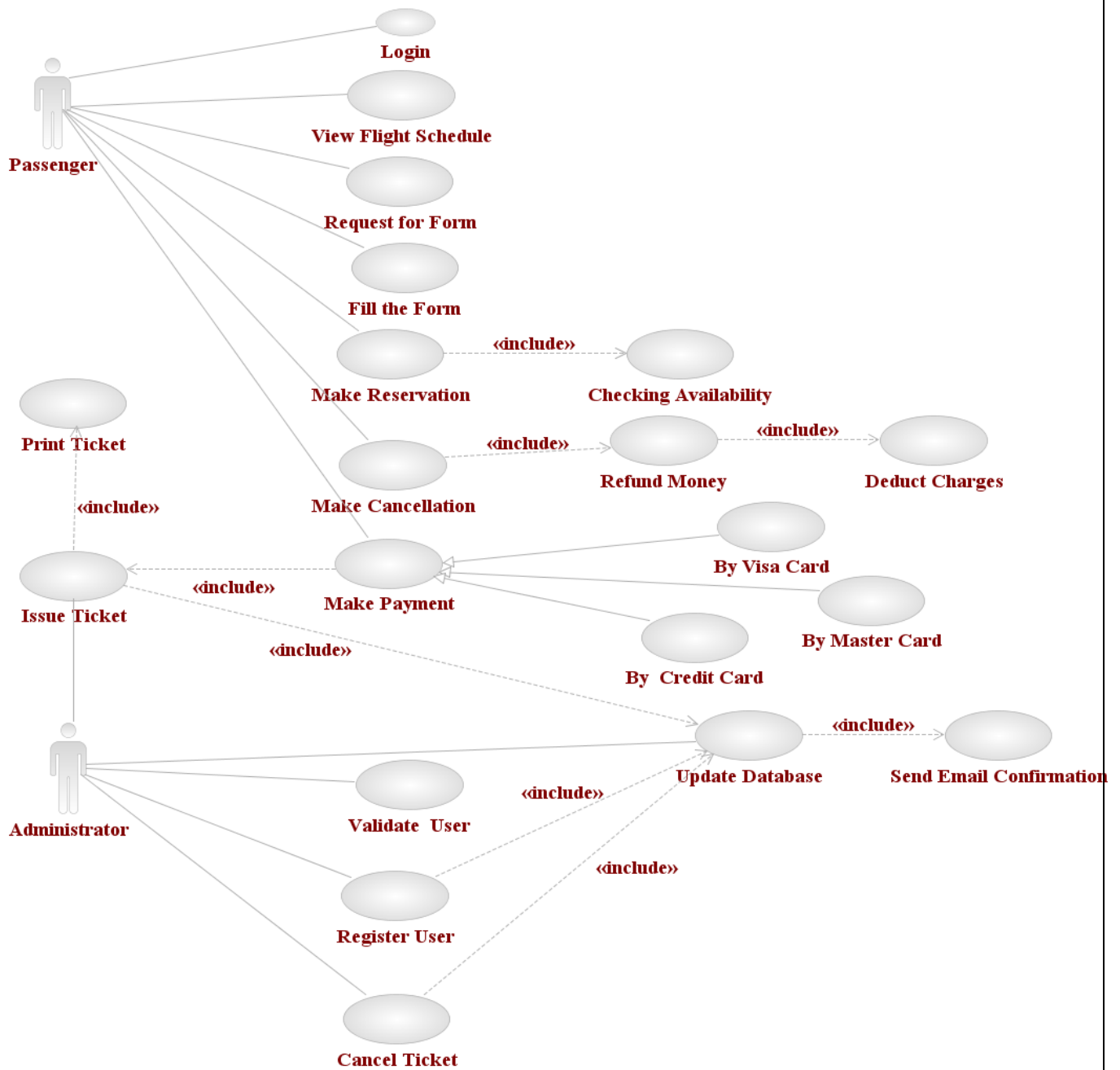
■ Example: Use Case Diagram for Hostel Management System



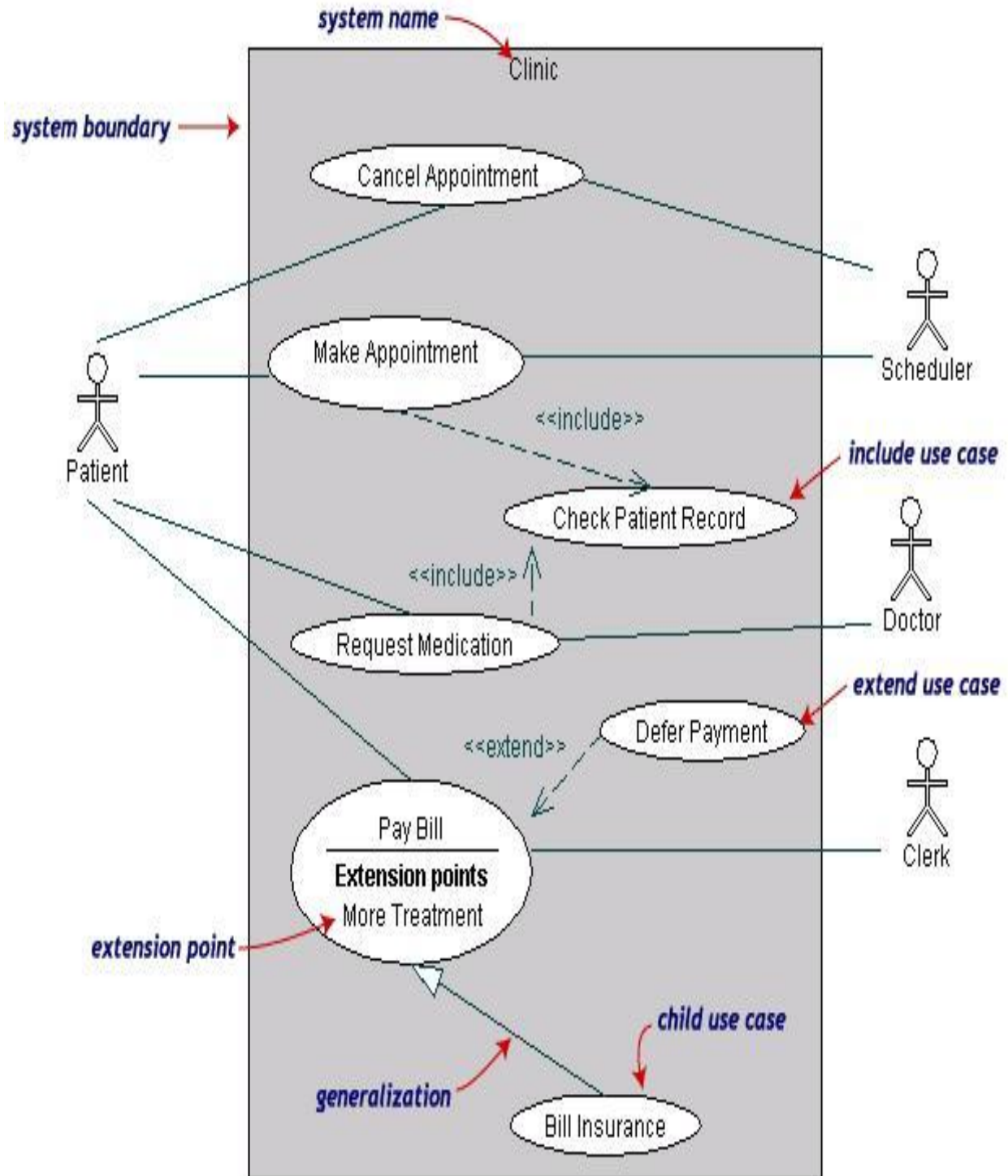
Example: Use Case Diagram for Banking System



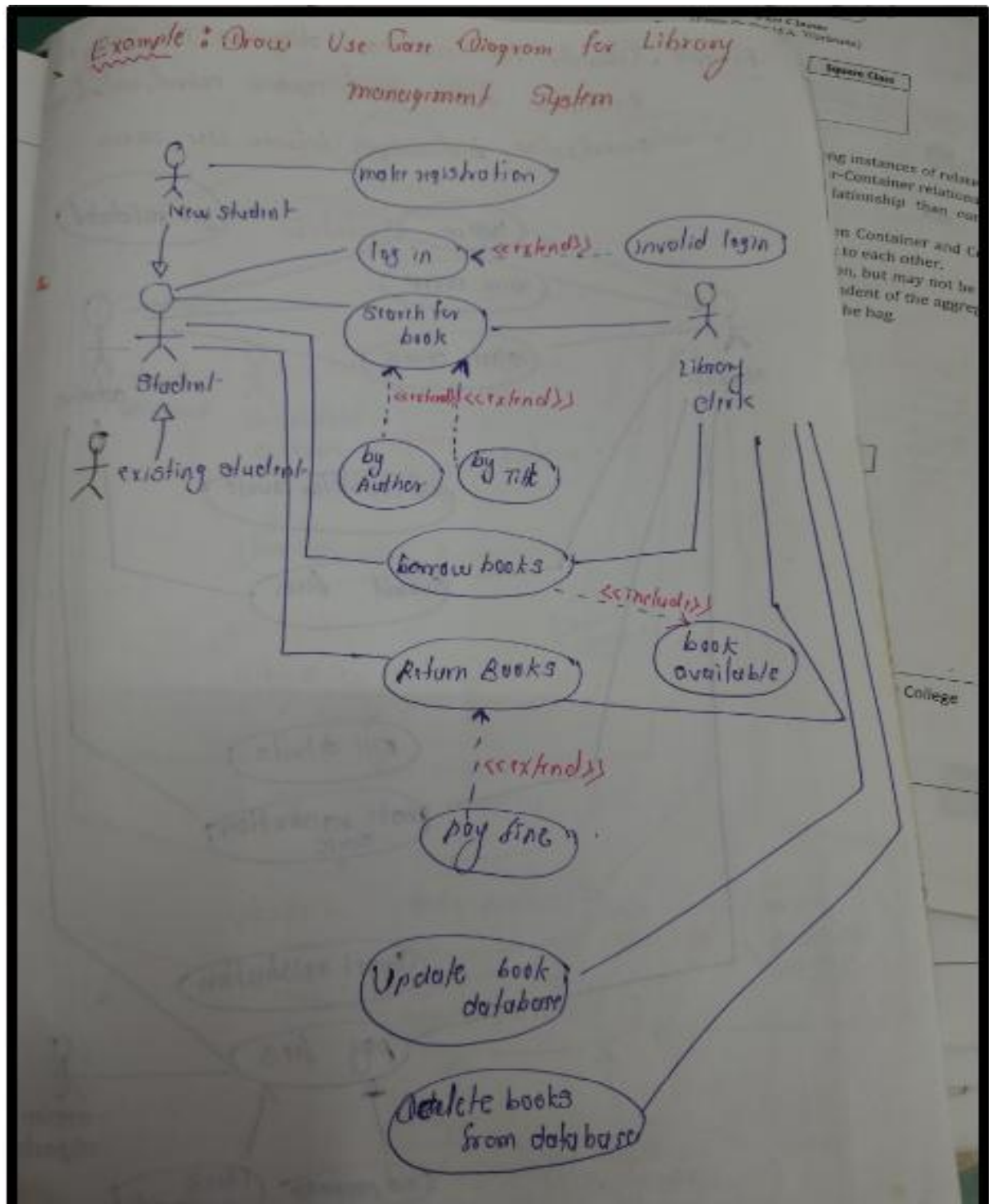
Example: Use Case Diagram for Online Airline Reservation System



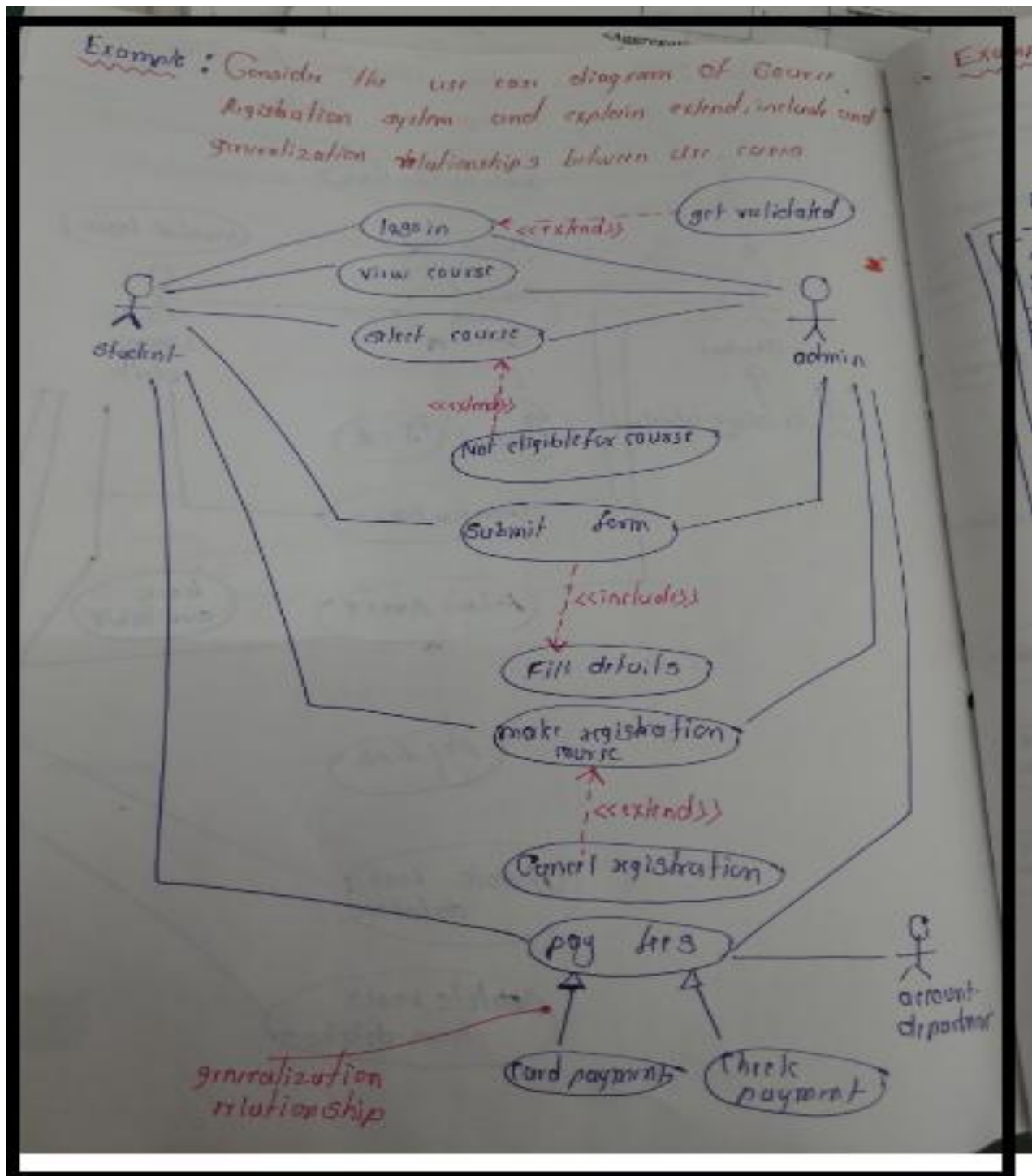
Example: Use Case Diagram for Hospital Management System

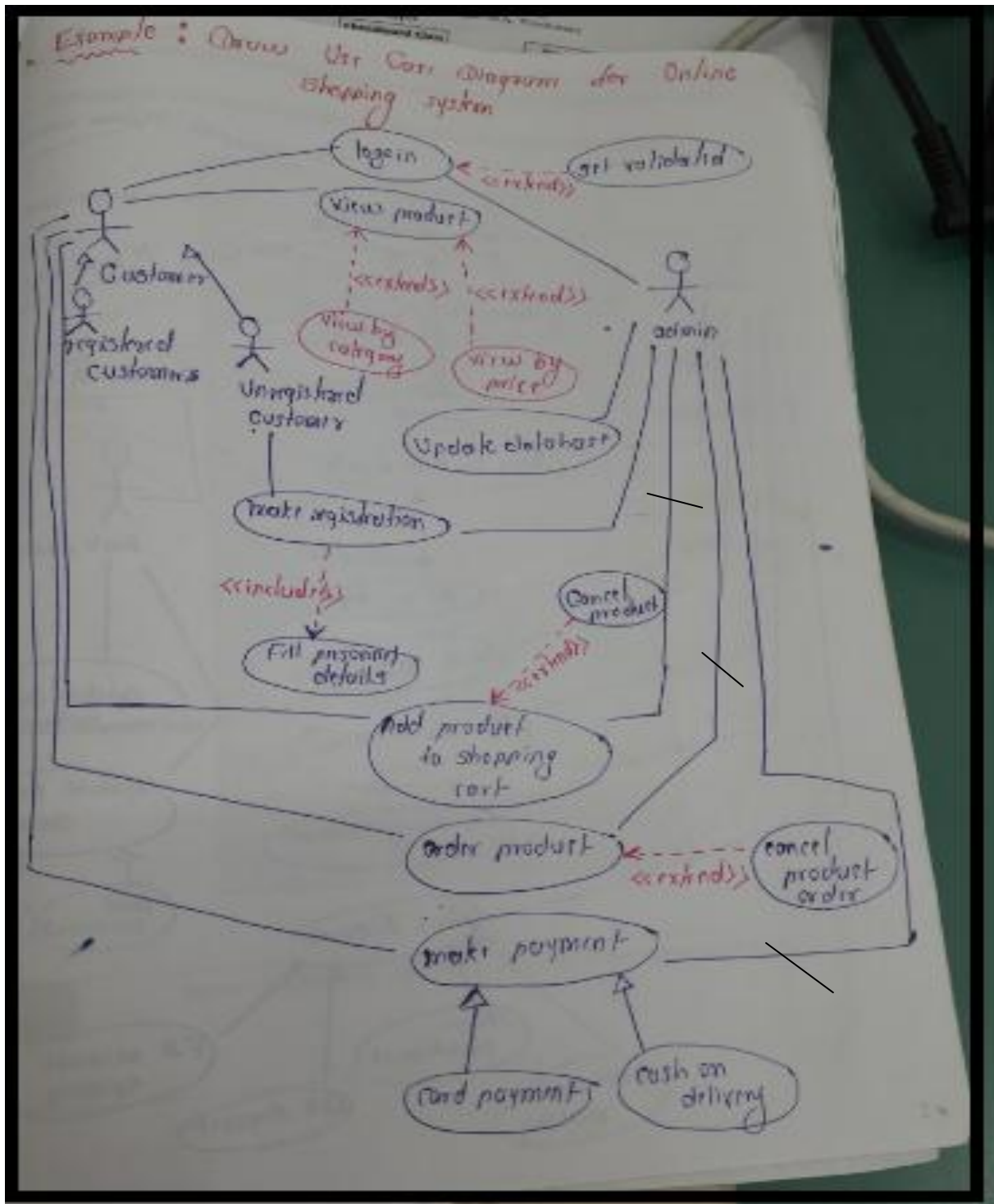


■ Example: Library Management system

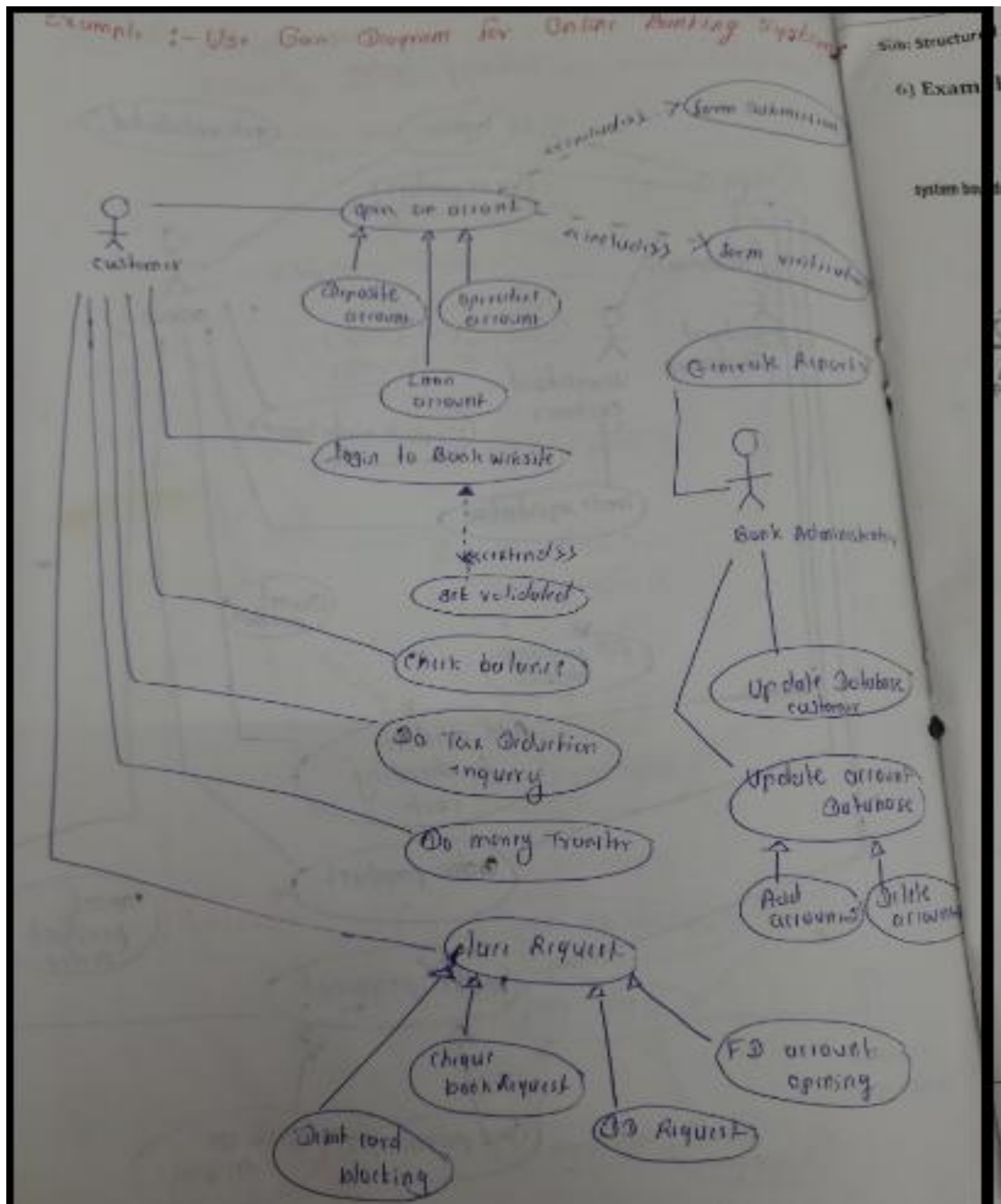


■ Example: Online Course Registration system

**Example:** Online shopping system



Example: Online banking system



■ Class Diagram

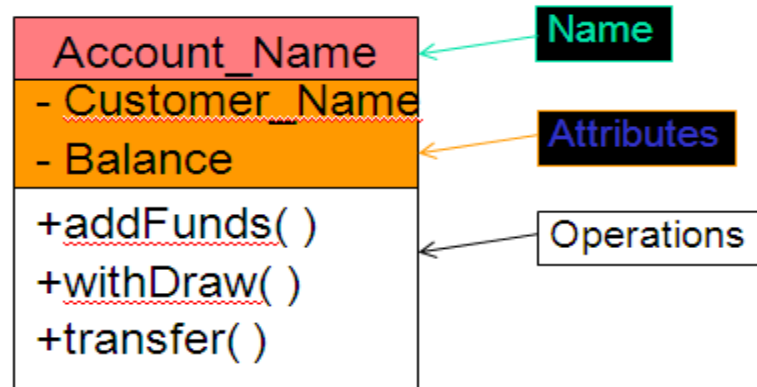
- Class diagram is a collection of static elements **such as classes and their relationships connected as a graph to each other.**
- It shows the dependency between the classes that can be used in our system.
- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- **Detailed class diagrams are used for developers**
- The Class Diagram shows a **set of classes, interfaces, and collaborations** and their relationships.
- **There is most common diagram in modeling the object oriented systems and are used to give the static view of a system.**
- UML Class Diagram consists of;

+ Classes

+ Relationships

+ Class:

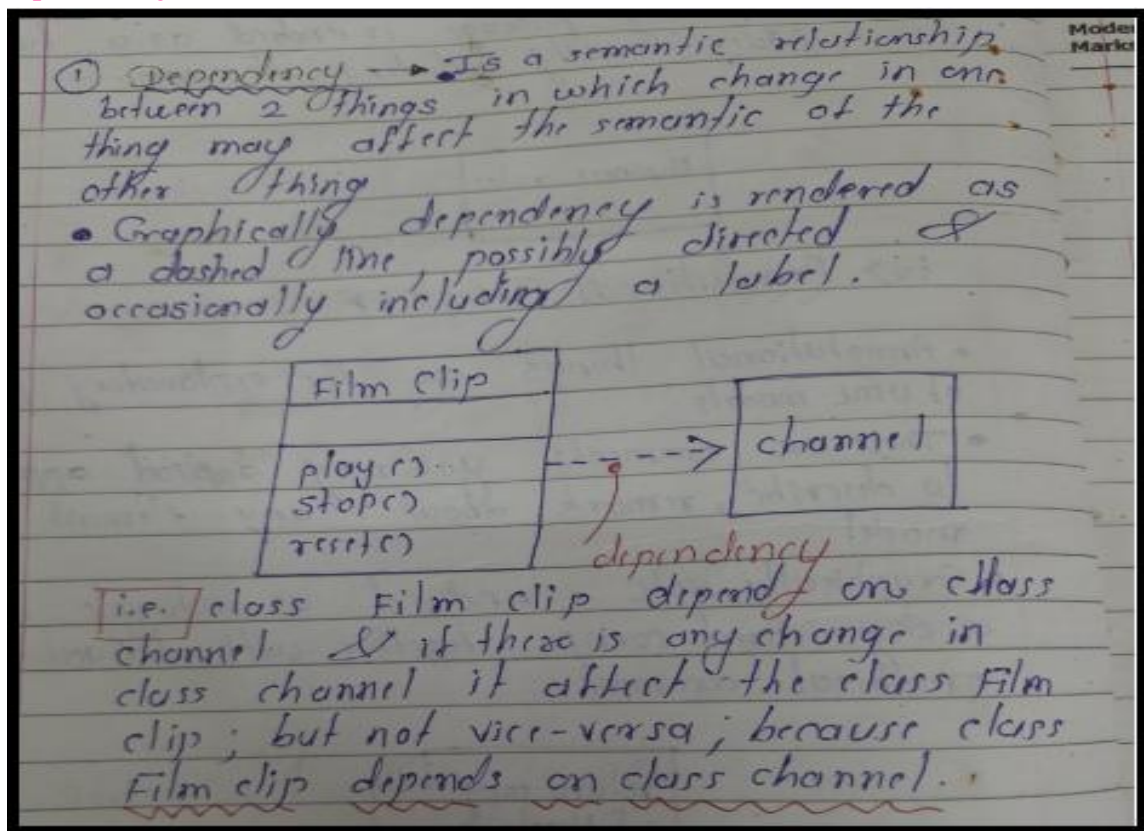
- **Each class is represented by a rectangle subdivided into three compartments**
 - Name
 - Attributes
 - Operations
- Modifiers are used to indicate visibility of attributes and operations.
- '+' is used to denote *Public* visibility (everyone)
- '#' is used to denote *Protected* visibility (friends and derived)
- '-' is used to denote *Private* visibility (no one)



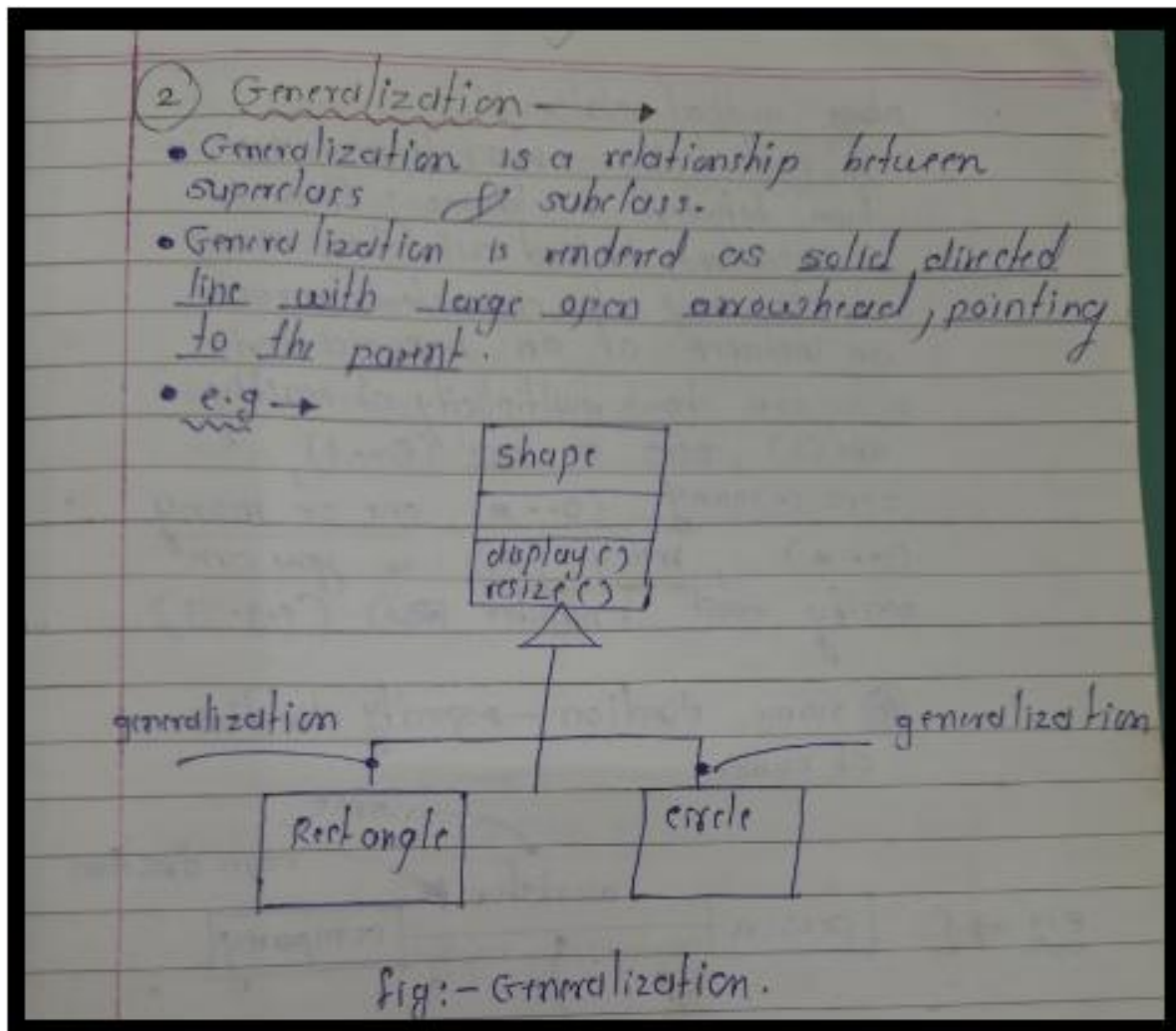
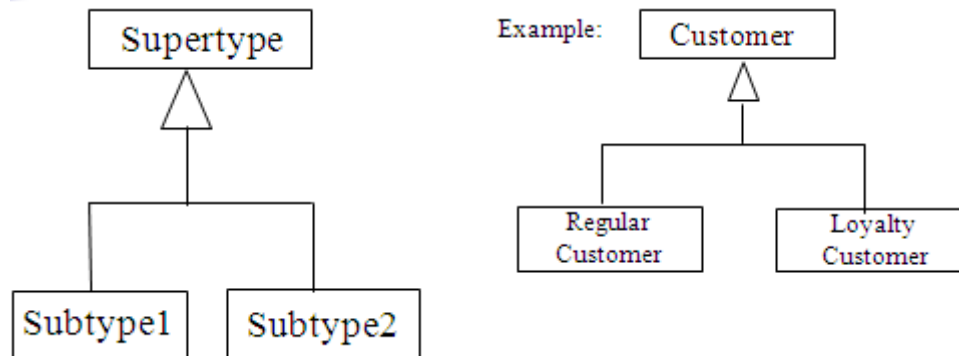
Relationships:

- There are three kinds of Relationships
- **Generalization** (parent-child relationship)
- **Association** (student enrolls in course)
- **Dependency** ----->

1. Dependency

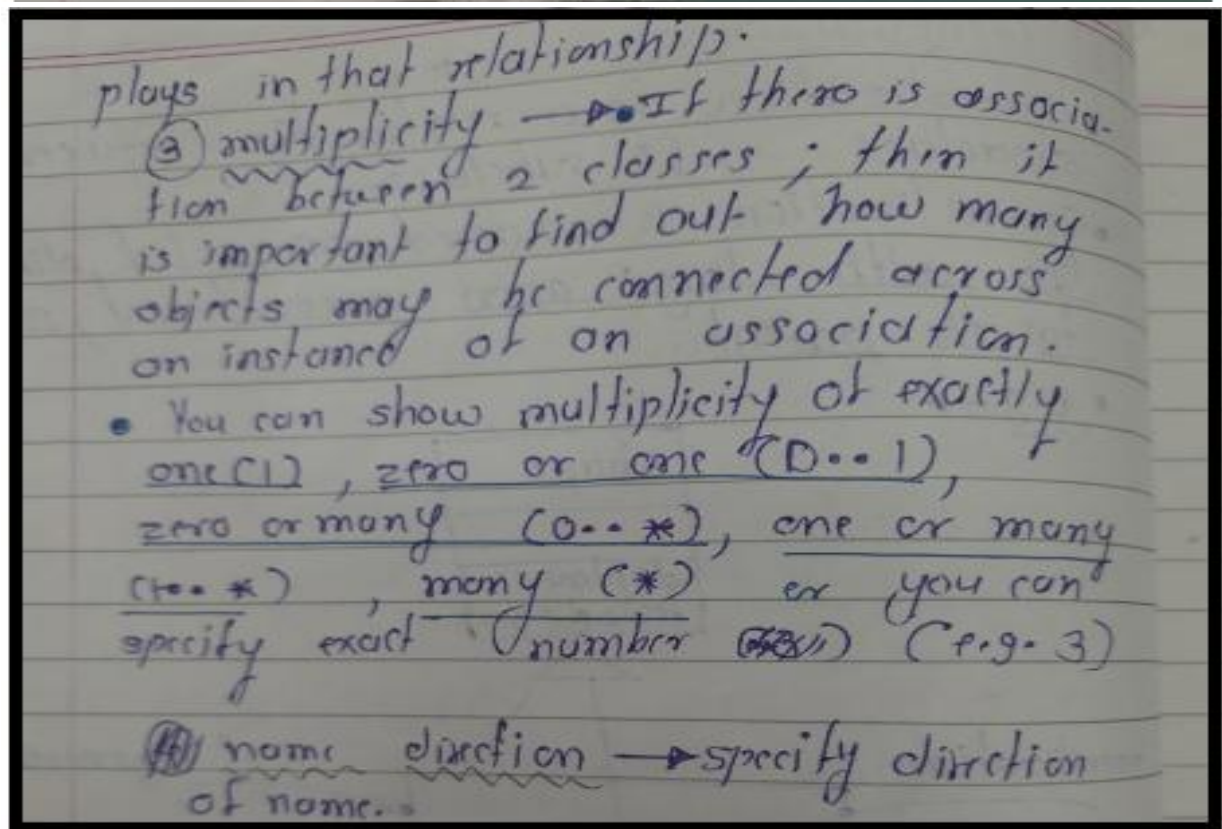
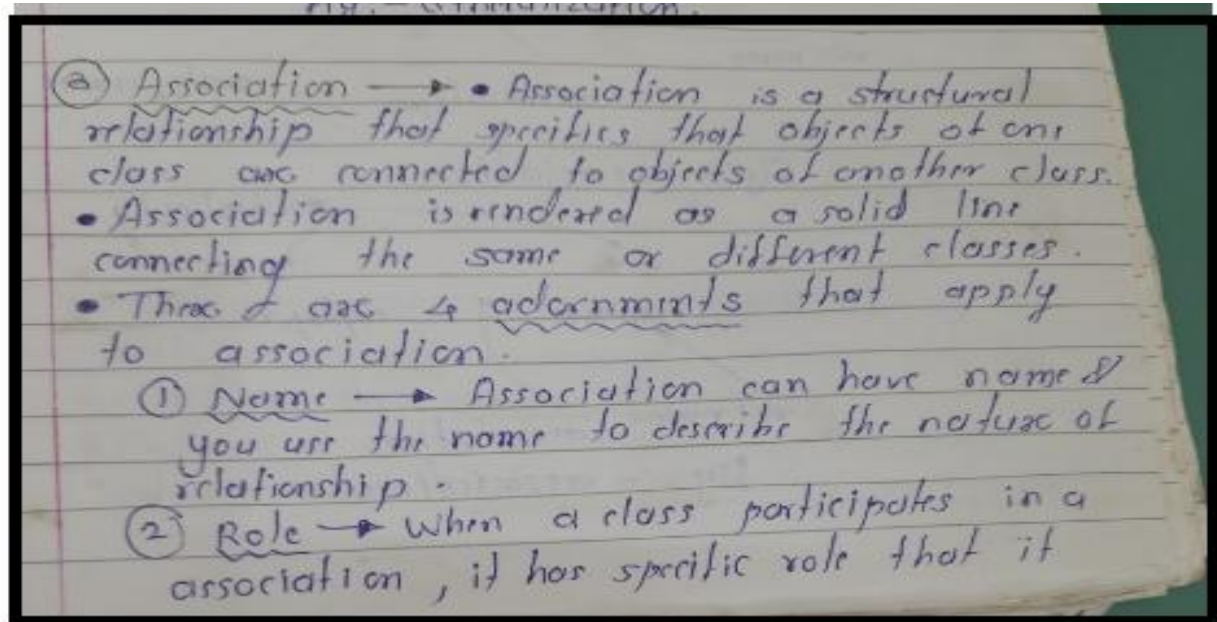


2. **Generalization:** Generalization expresses a parent/child relationship among related classes.



3. Association:

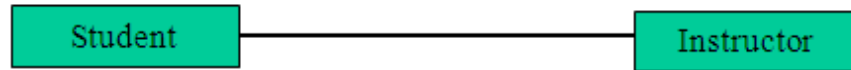
Represent relationship between instances of classes



Example 1:

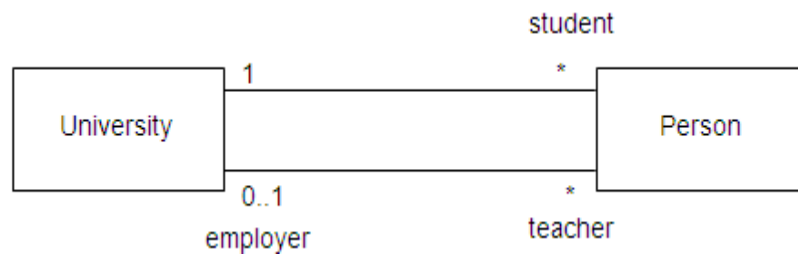
- ✓ Student enrolls in a course

- ✓ Courses have students
- ✓ Courses have exams
- ✓ Etc.

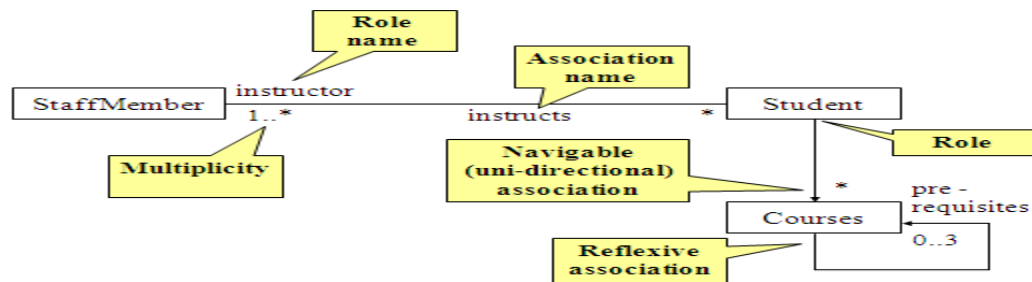


Example 2:

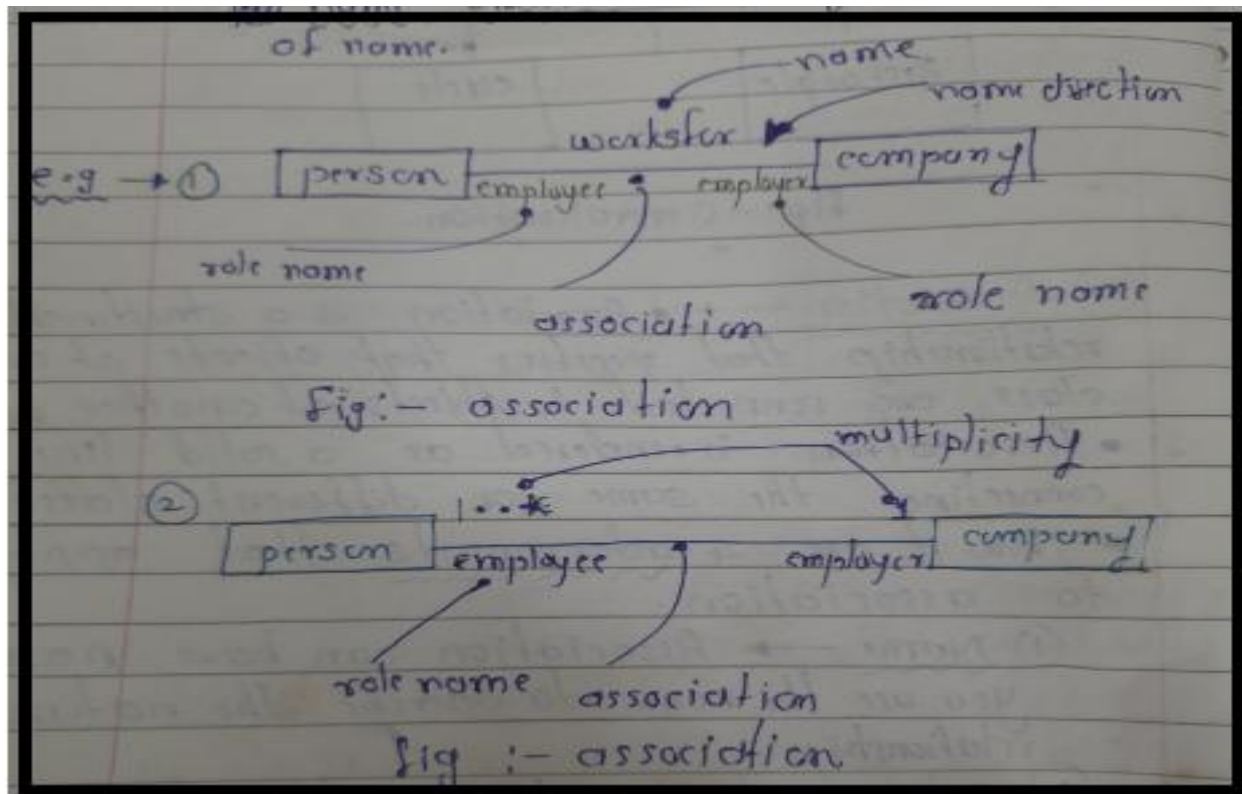
- ✓ Role names (e.g. enrolls)
- ✓ Multiplicity (e.g. One course can have many students)
- ✓ Navigability (unidirectional, bidirectional)
- ✓ Example:



Example 3:



Example 4:

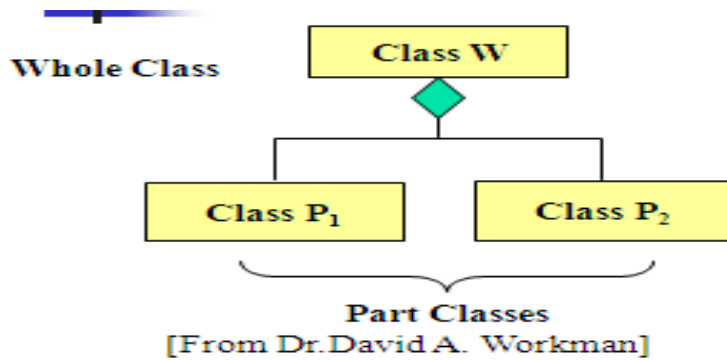


❖ Associations can be further classified as

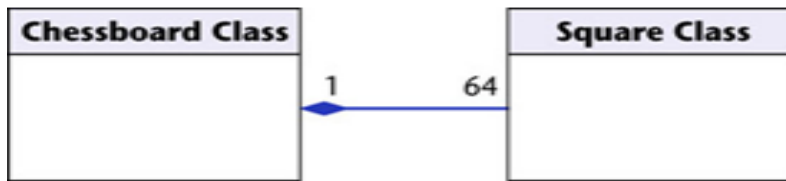
- Aggregation
- Composition

➤ **Composition:**

- ✓ **Composition** is really a strong form of **association**
- ✓ components have only one owner
- ✓ components cannot exist independent of their owner
- ✓ components live or die with their owner
- ✓ e.g. Each car has an engine that cannot be shared with other cars.
- ✓ **Example:** A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.

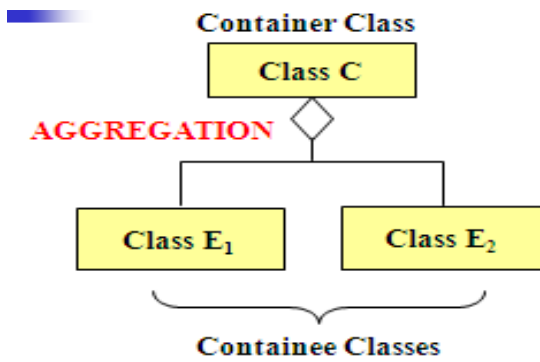


Example

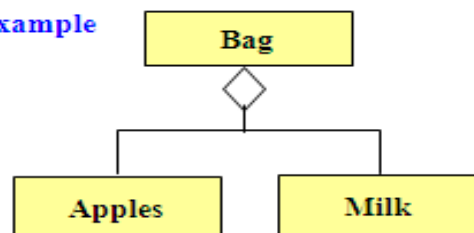


➤ **Aggregations:**

- ✓ Expresses a relationship among instances of related classes. It is a specific kind of **Container-Container relationship**.
- ✓ Express a more informal relationship than composition expresses.
- ✓ Aggregation is appropriate when Container and Containers have no special access privileges to each other.
- ✓ May form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. E.g. Apples may exist independent of the bag.
- ✓ **Example:**



Example



Example:

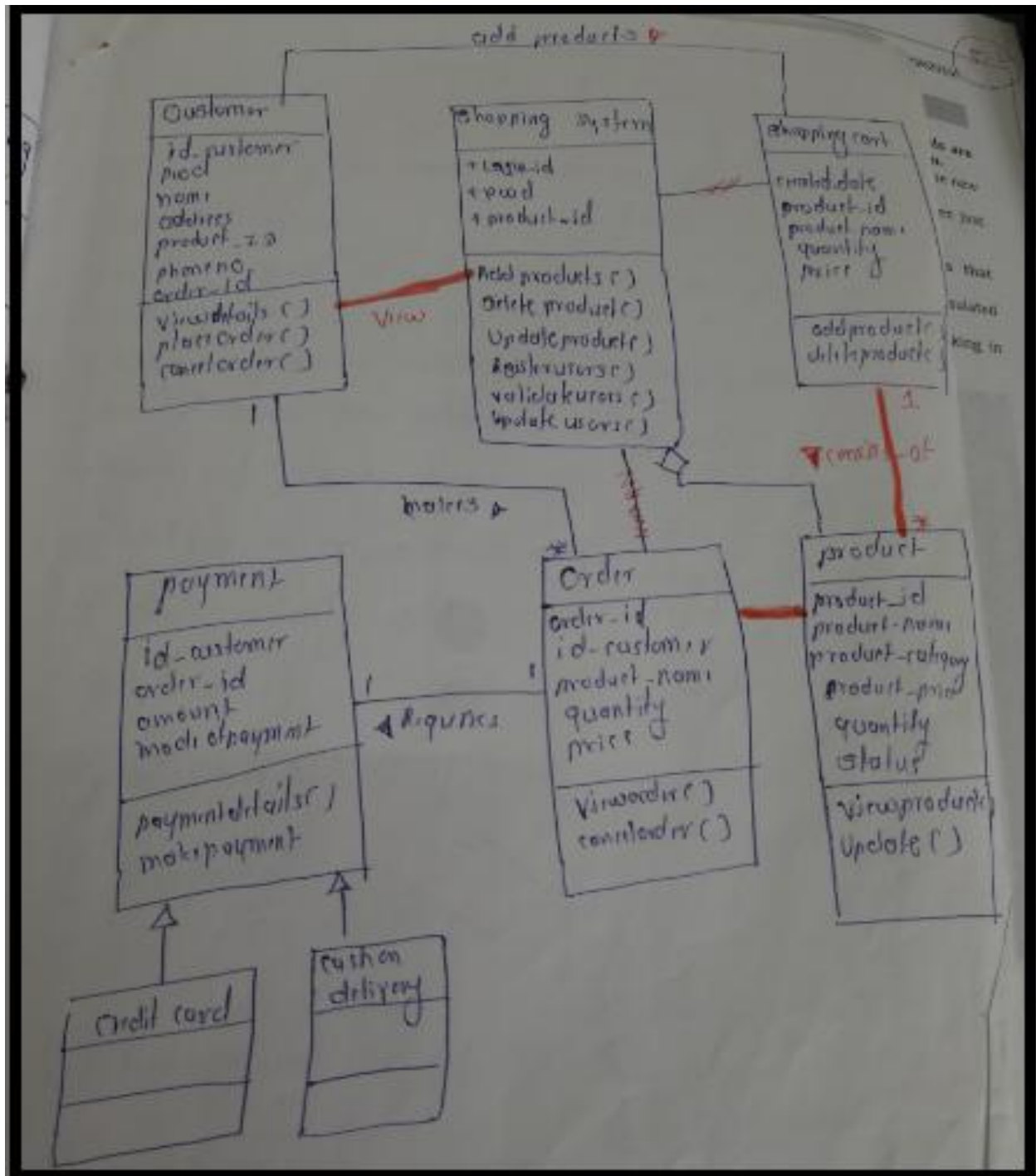


Figure: Online Shopping System

Example:

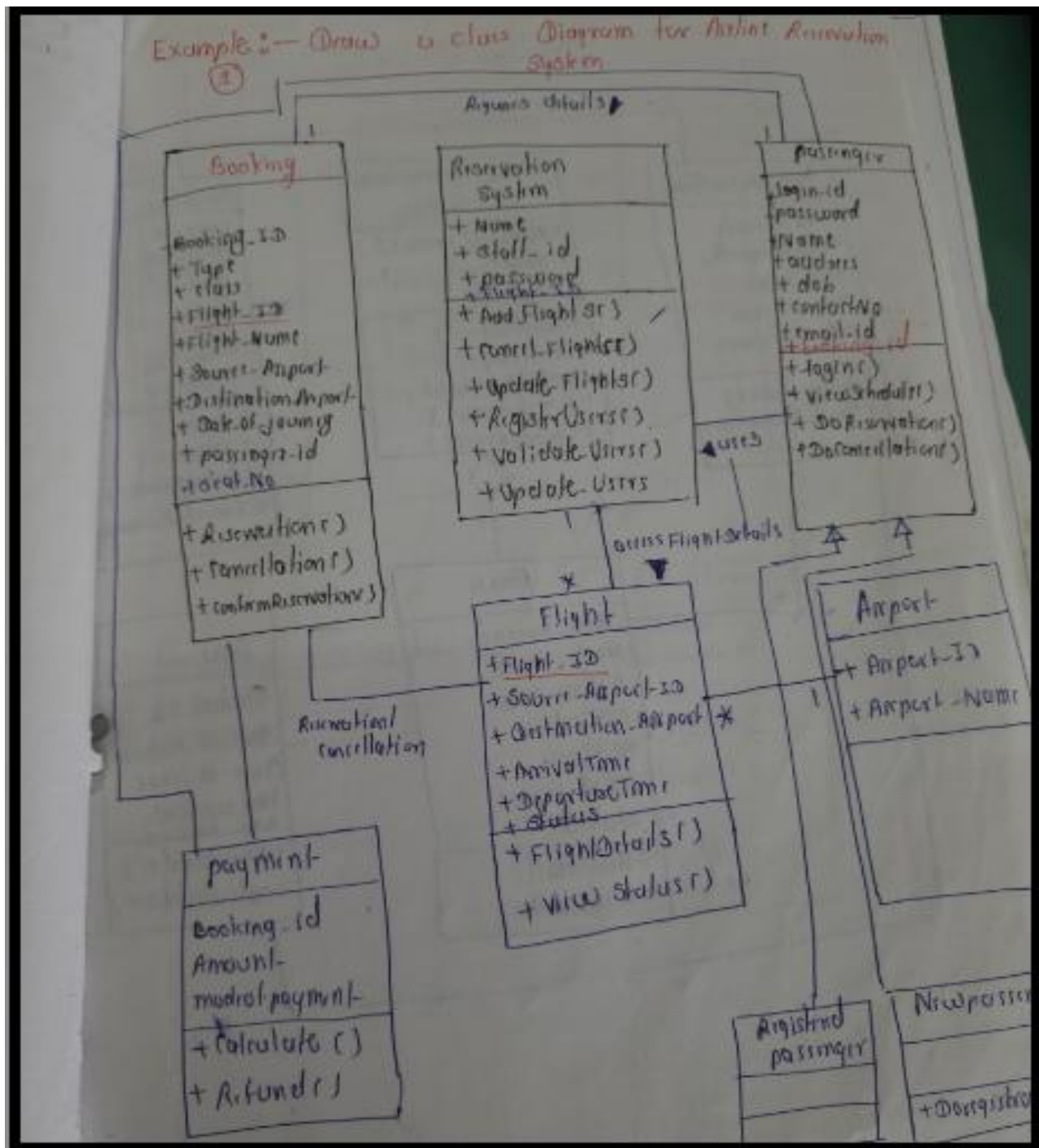


Figure: Airline Reservation System

Example

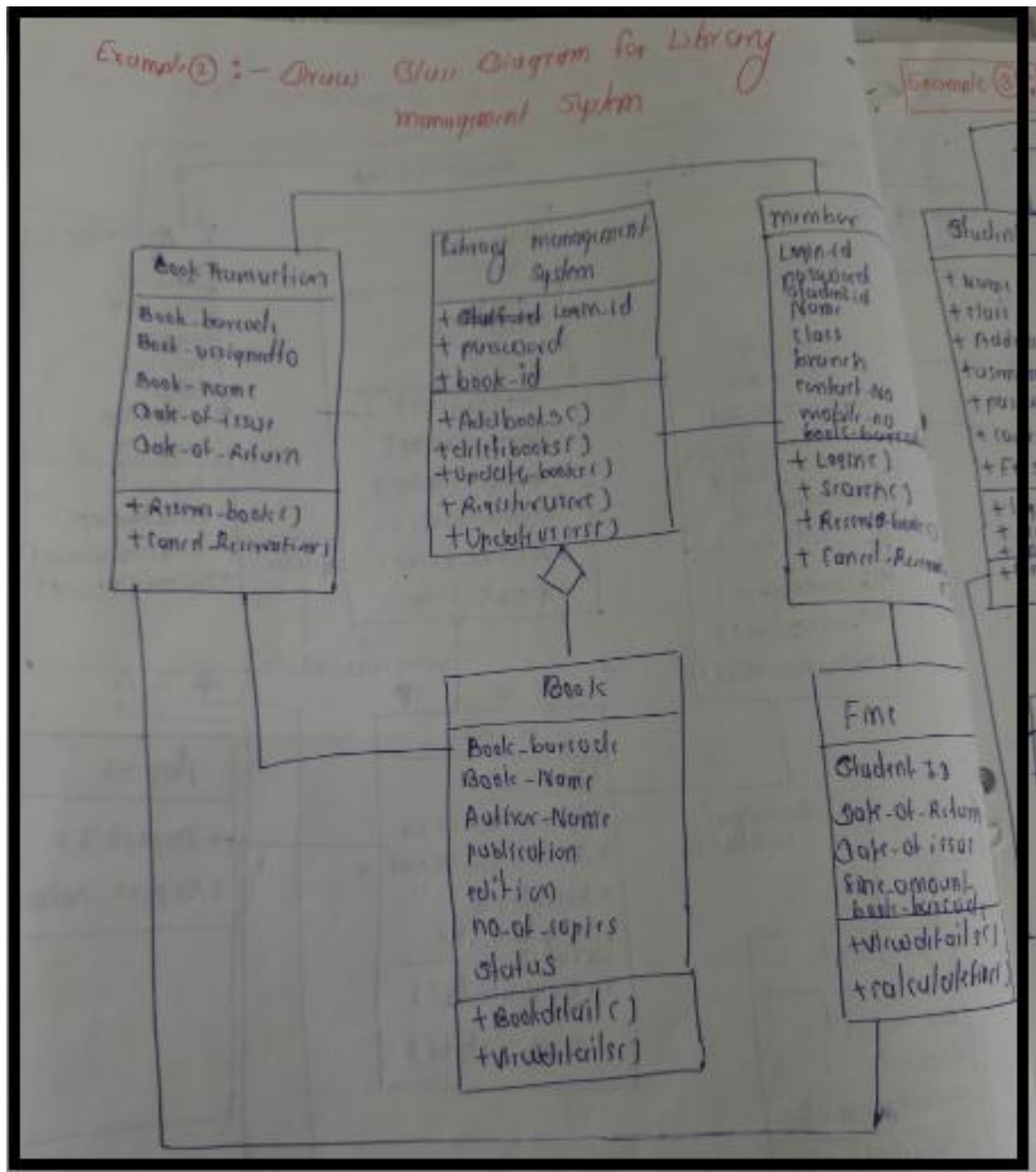


Figure: Library Management System

Example

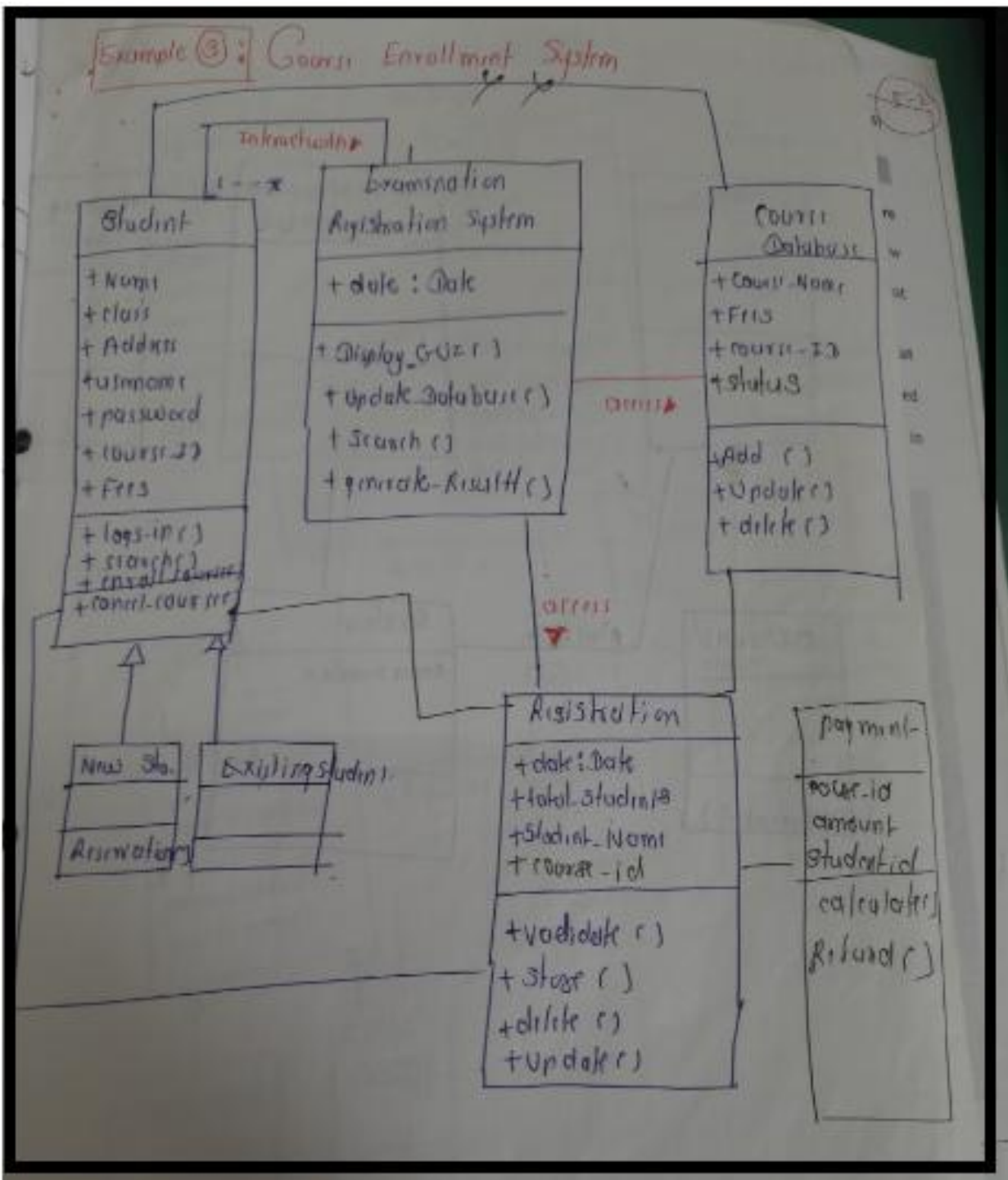


Figure: Online Course Enrollment system

■ Interaction Diagram

- Show how objects interact with one another, it shows an interaction of set of objects and their relationships, including the messages that they exchanged among them. These set of messages are used by the objects to communicate with each other.
- **UML supports two types of interaction diagrams**

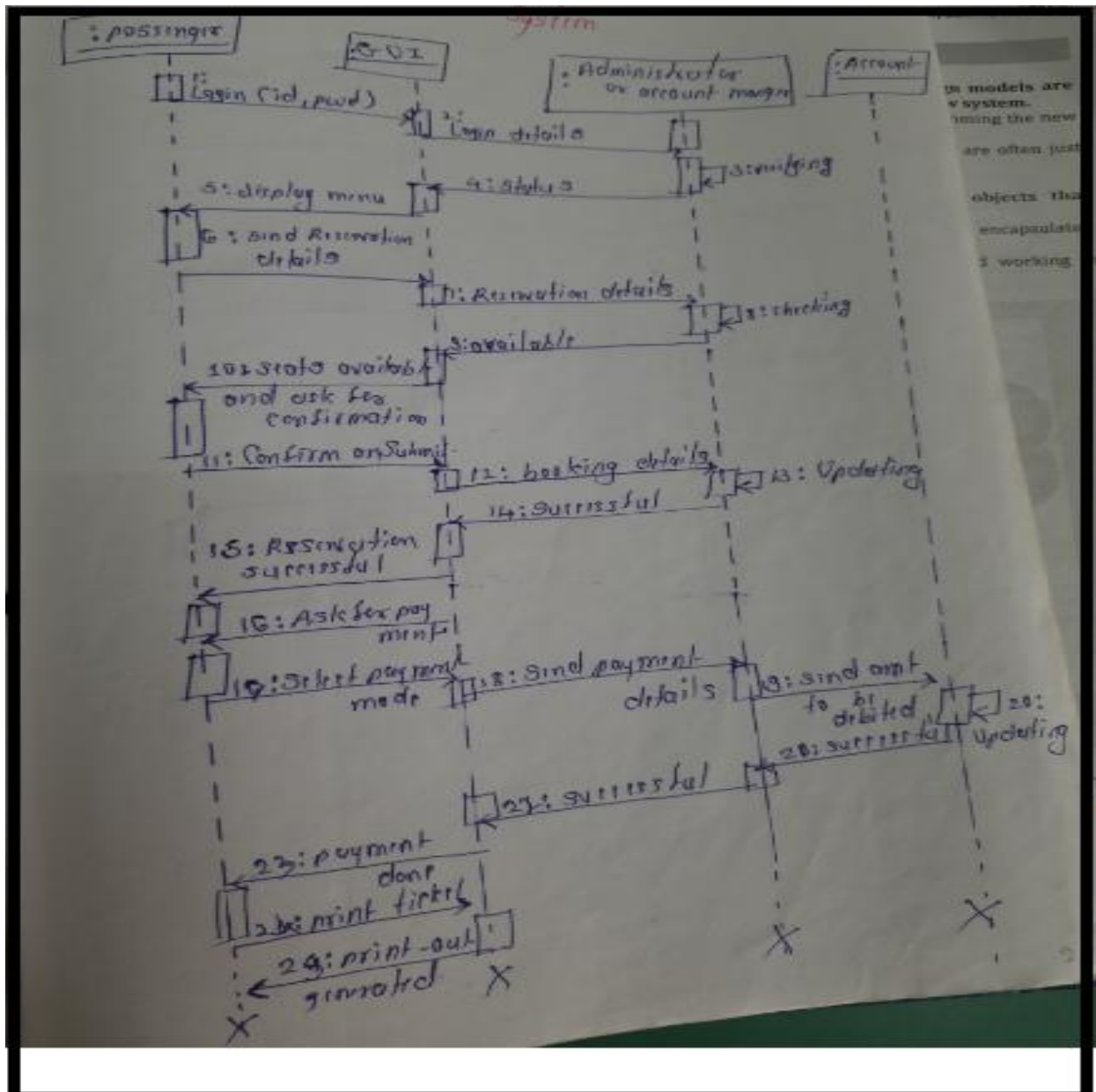
-  Sequence diagrams

-  Collaboration diagrams

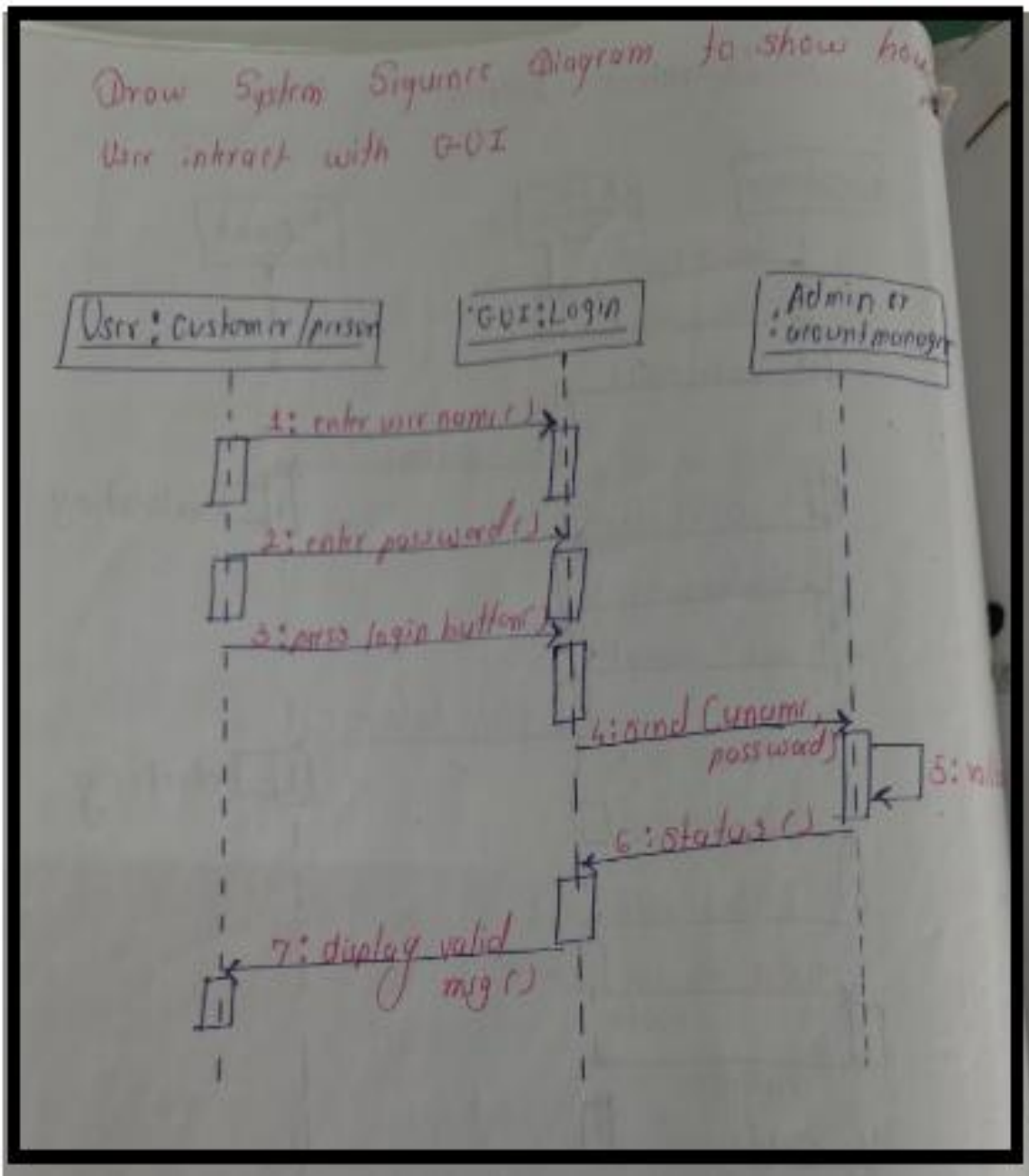
Sequence Diagram

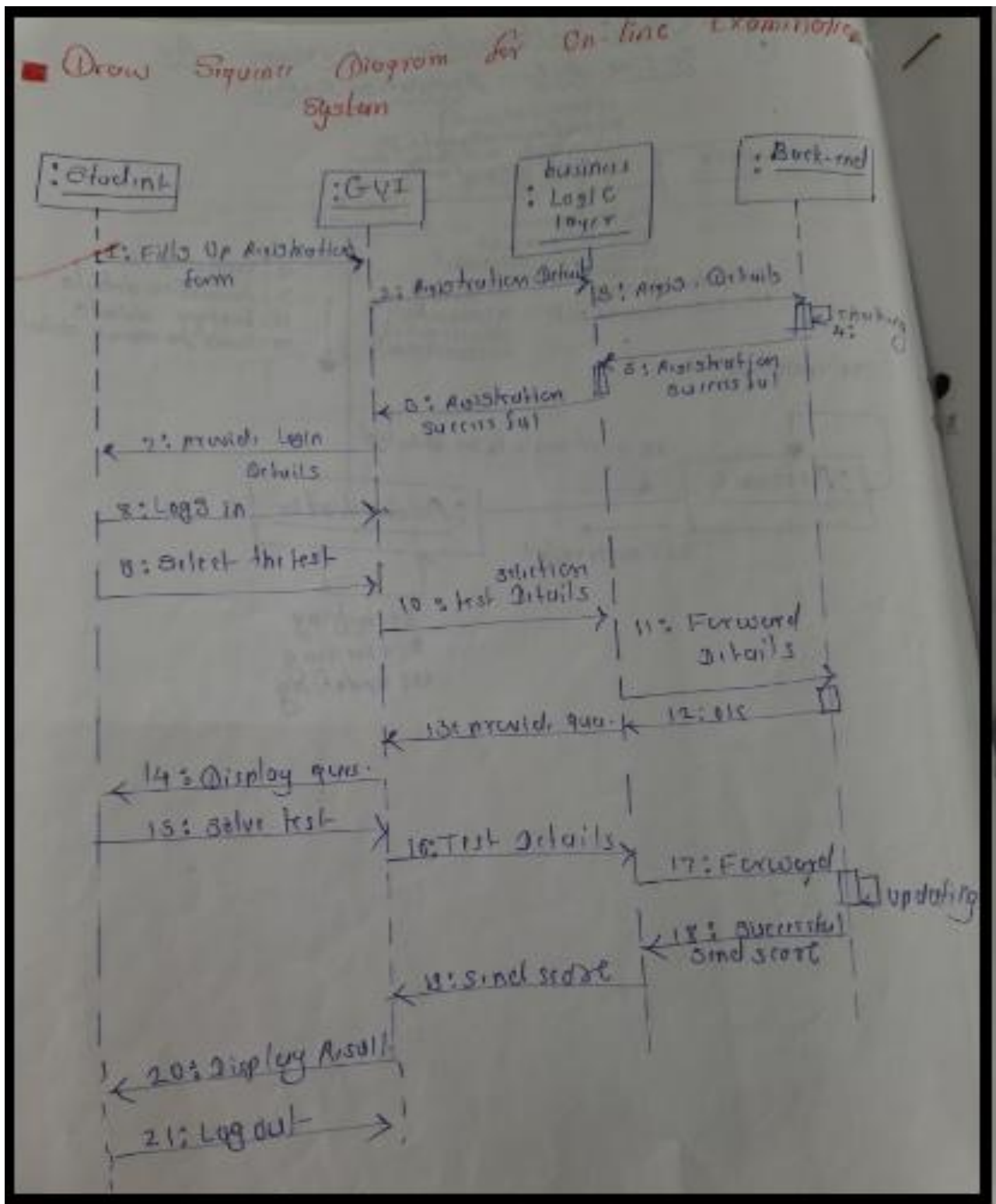
- Sequence diagram shows an interaction arranged in a time sequence. It is an alternate way to understand the overall flow of the control of the system program.
- **UML Sequence Diagram consists of;**
 - ✓ **Objects**
 - ✓ **Object life line**
 - ✓ **Focus of Control**
 - ✓ **Messages send** by the objects to communicate with each other
- Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.
- Sequence diagram shows the sequence of events occurs.
- The horizontal dimension shows the objects participating in the interaction.
- The vertical arrangement of messages indicates their order.

Example: Online Airline Reservation system

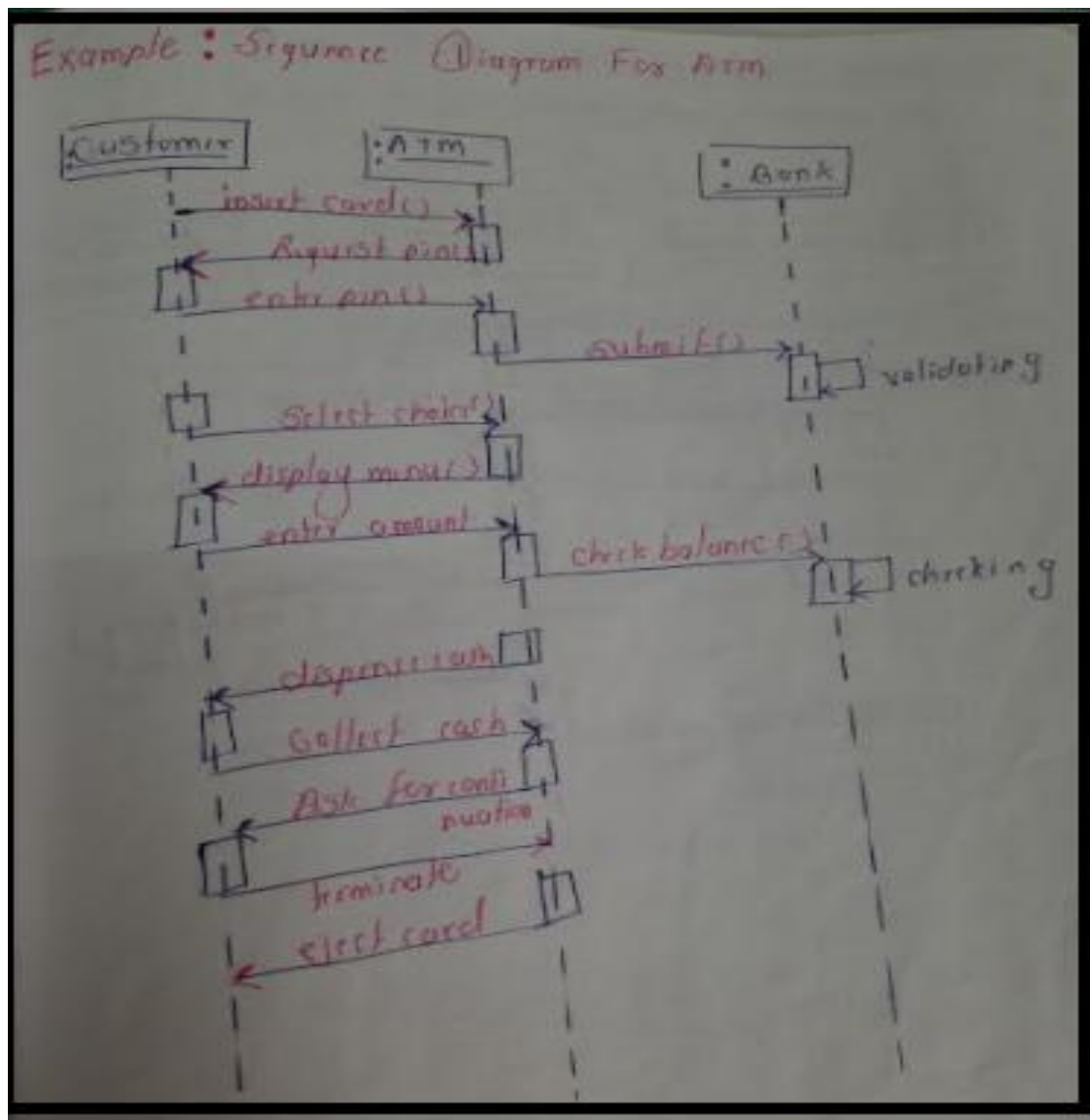


Example: How user interact with GUI

**Example: Online Examination System**



Example: Sequence Diagram for ATM system



✚ Collaboration Diagram:

- Collaboration diagrams are equivalent to sequence diagrams. All the features of sequence diagrams are equally applicable to collaboration diagrams
- Use a sequence diagram when the transfer of information is the focus of attention.
- Use a collaboration diagram when concentrating on the classes.

① Collaboration diagram emphasizes the organization of the objects that participate in an interaction.

② Collaboration diagram consists of,
i) objects
ii) links

③ Collaboration diagrams have 2 features
i) path → path indicate how one object is linked to another.

• You can attach a path stereotype to the far end of a link such as;

① <<local>> → indicates that the object is local to the sender.

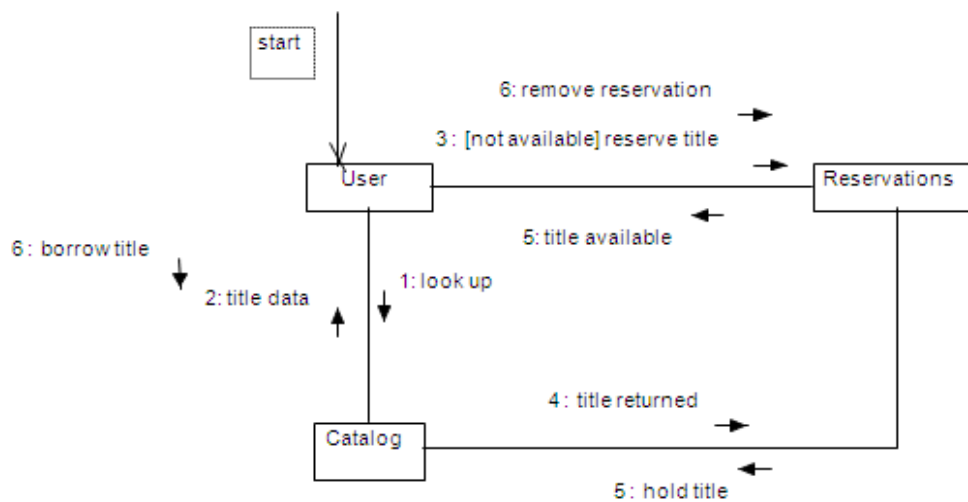
② <<global>> →

③ <<self>> →

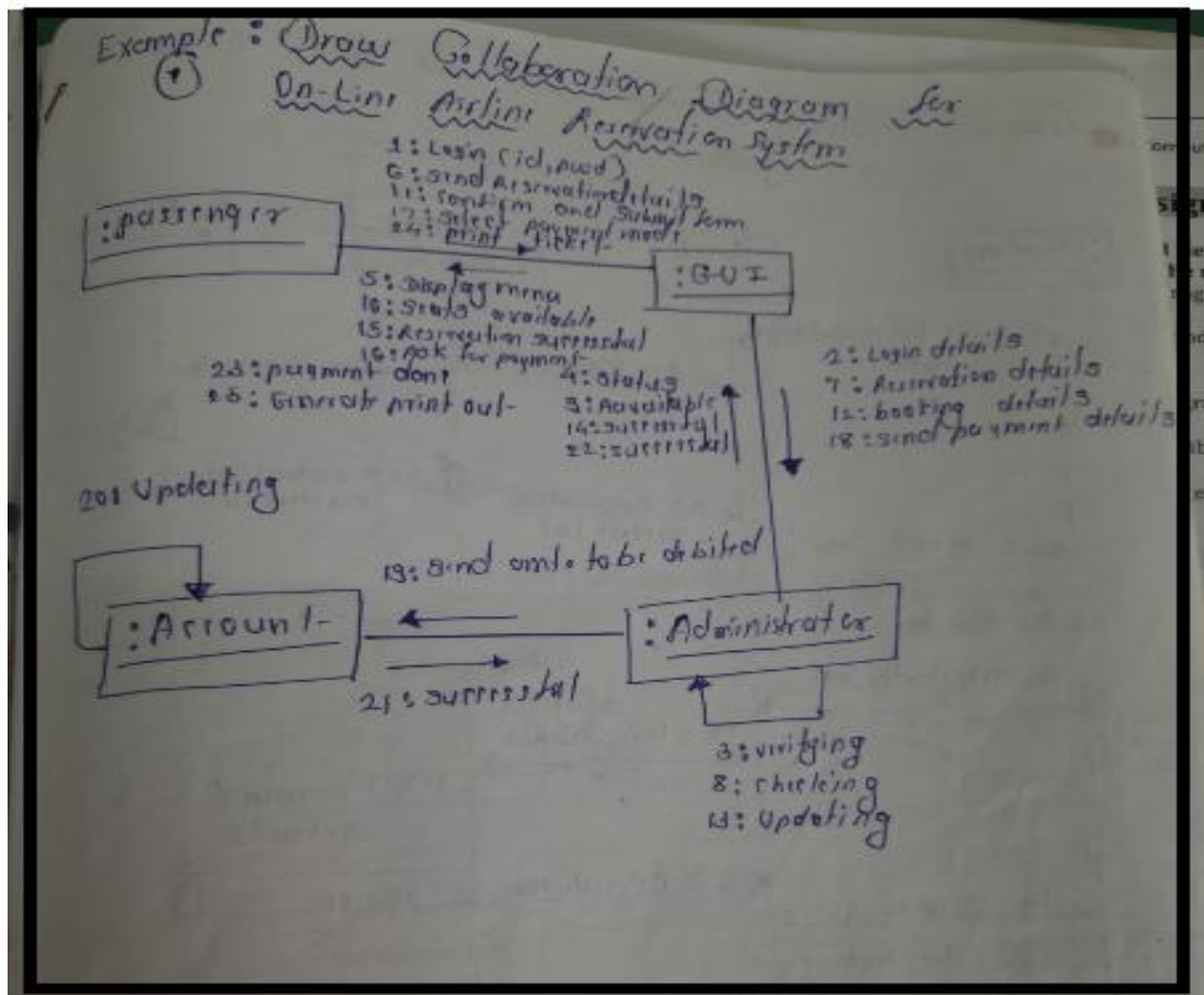
④ <<parameter>> →

ii) Sequence Number →

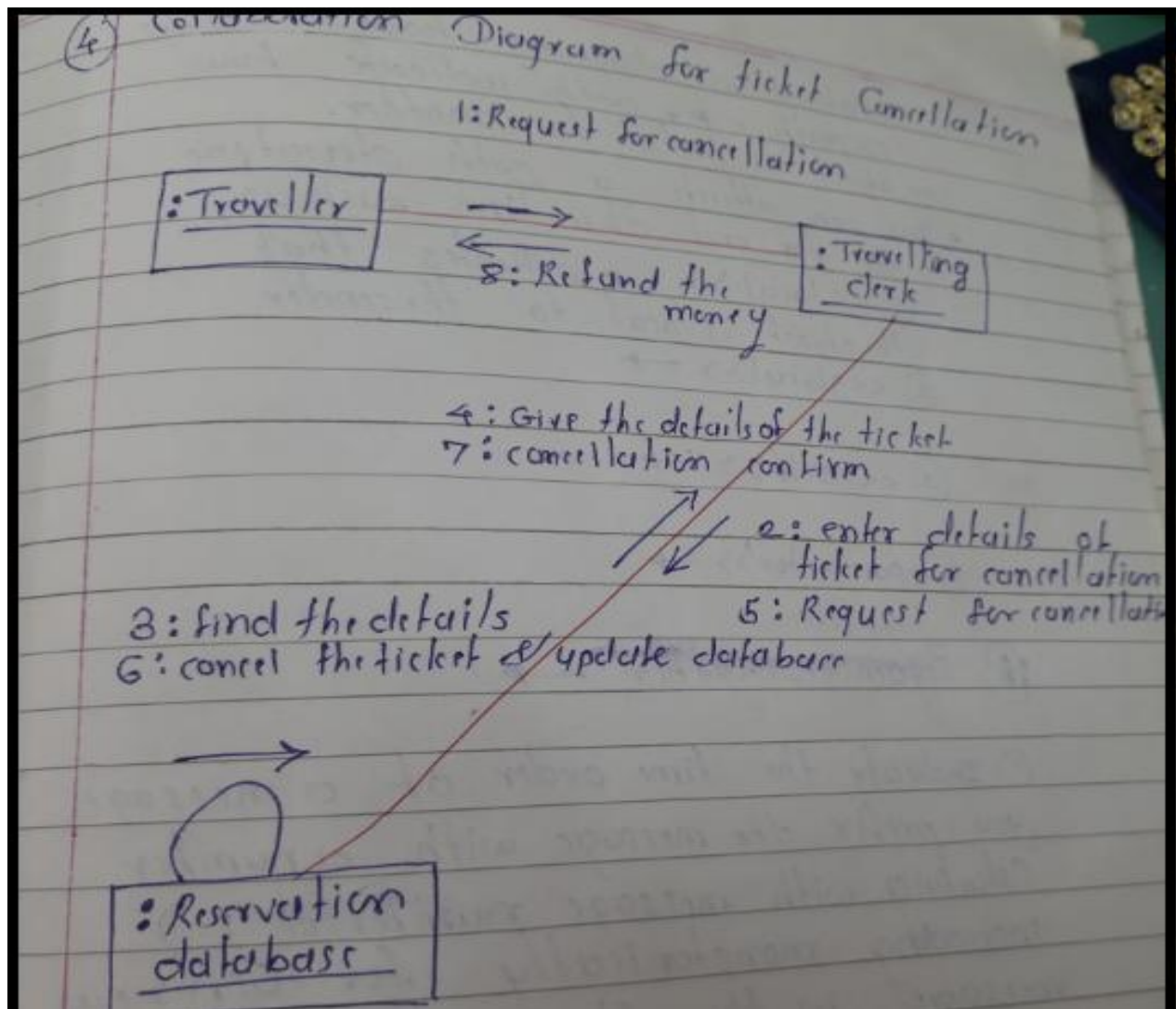
- ① Indicate the time order of a message, you prefix the message with a number (starting with message numbered 1) increasing monotonically for each new message in the flow of control 2, 3, --- & so on.
- ④ Collaboration diagram show the relationship between objects & the order of message passed between them.
- ⑤ → (arrow) indicates the message being passed between them.
- ⑥ There are many acceptable sequence-numbering schemes in UML.



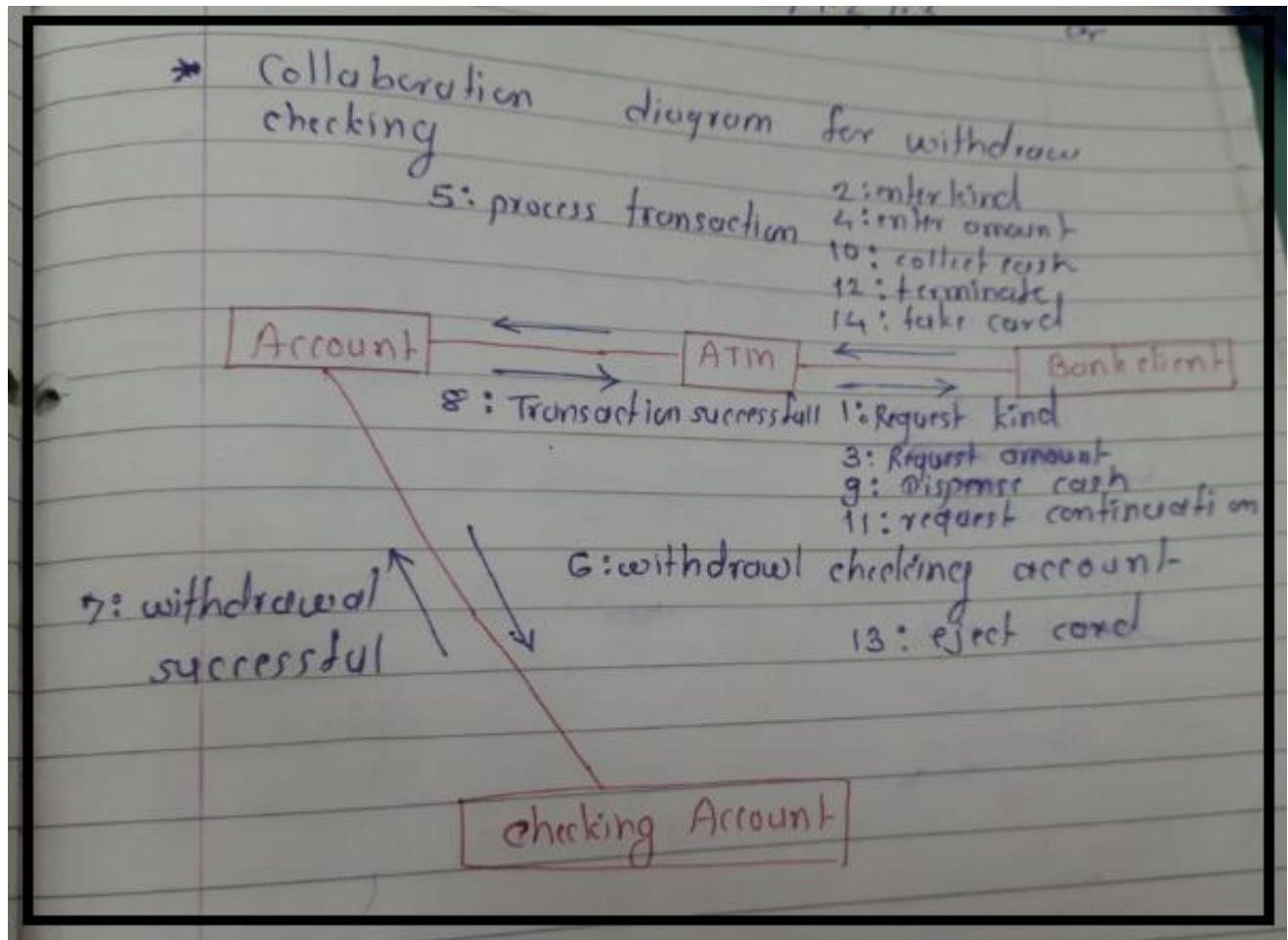
Example: Collaboration Diagram for Airline Reservation system



Example: Collaboration Diagram for Ticket Cancellation



Example: Withdrawal of Money using ATM



■ Activity Diagram

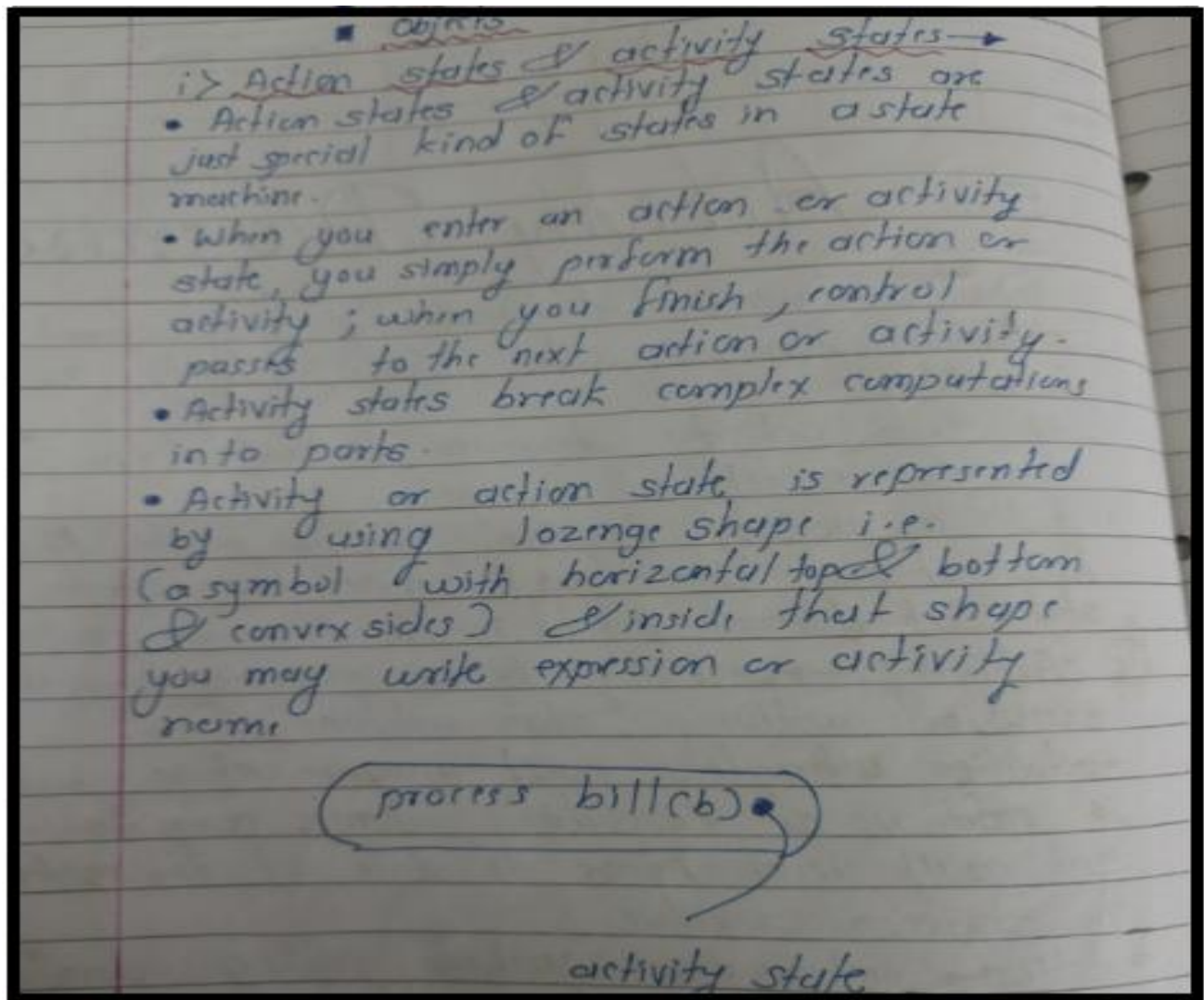
- Activity diagram is essentially a flowchart, showing flow of control from activity to activity.
- We use activity diagram to model dynamic flow of the system.
- Using activity diagram , we can model the flow of an object as it moves from state to state at different points in the flow of control.
- Activity Diagram includes following elements.

✓ Start activity and End Activity

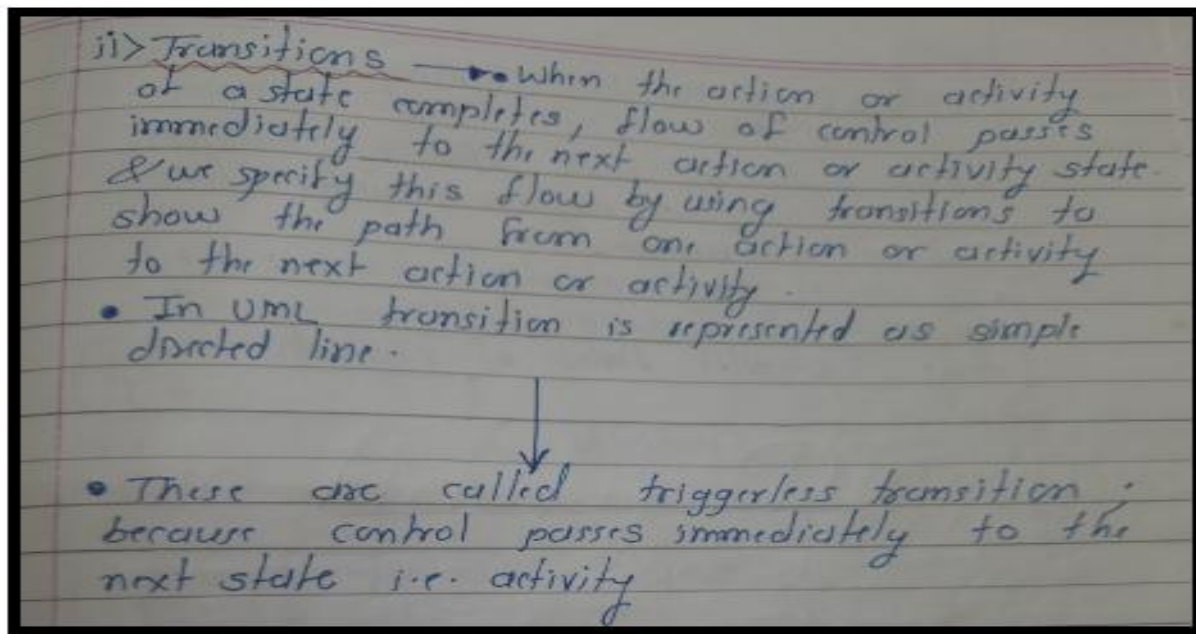
✓ Activity states and Action states

- ✓ Transitions
- ✓ Branching
- ✓ Forking and Joining
- ✓ Swim lane

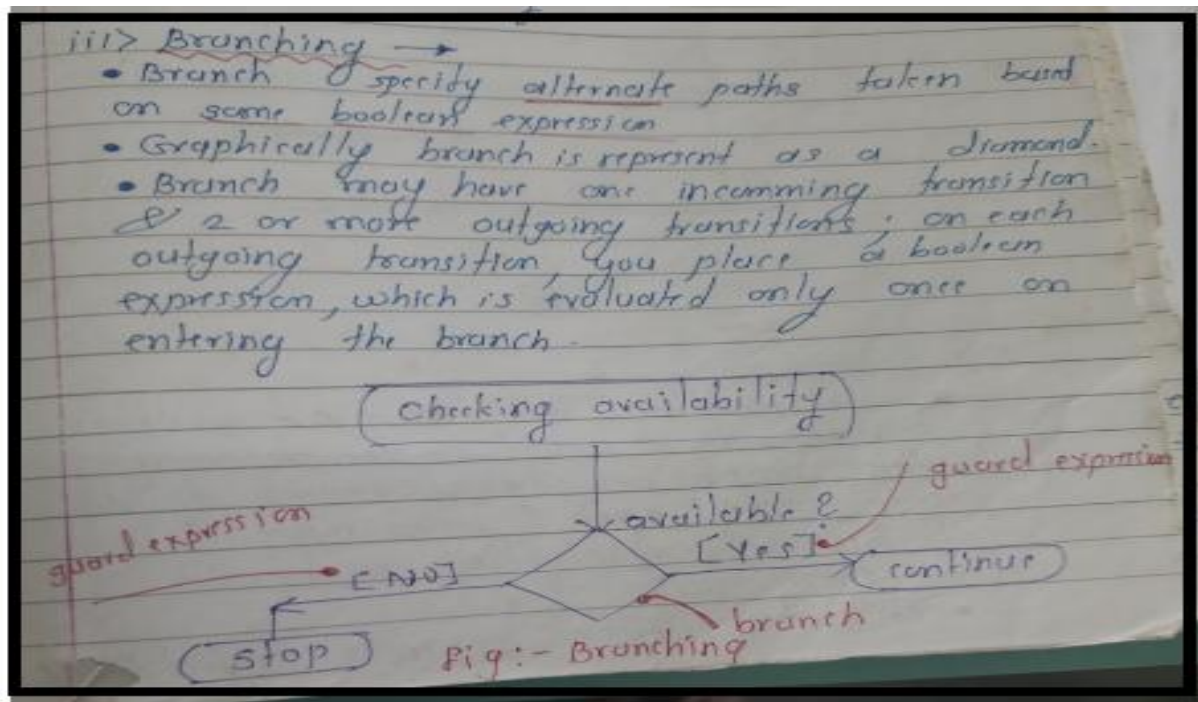
i) Activity and action states



ii) Transitions



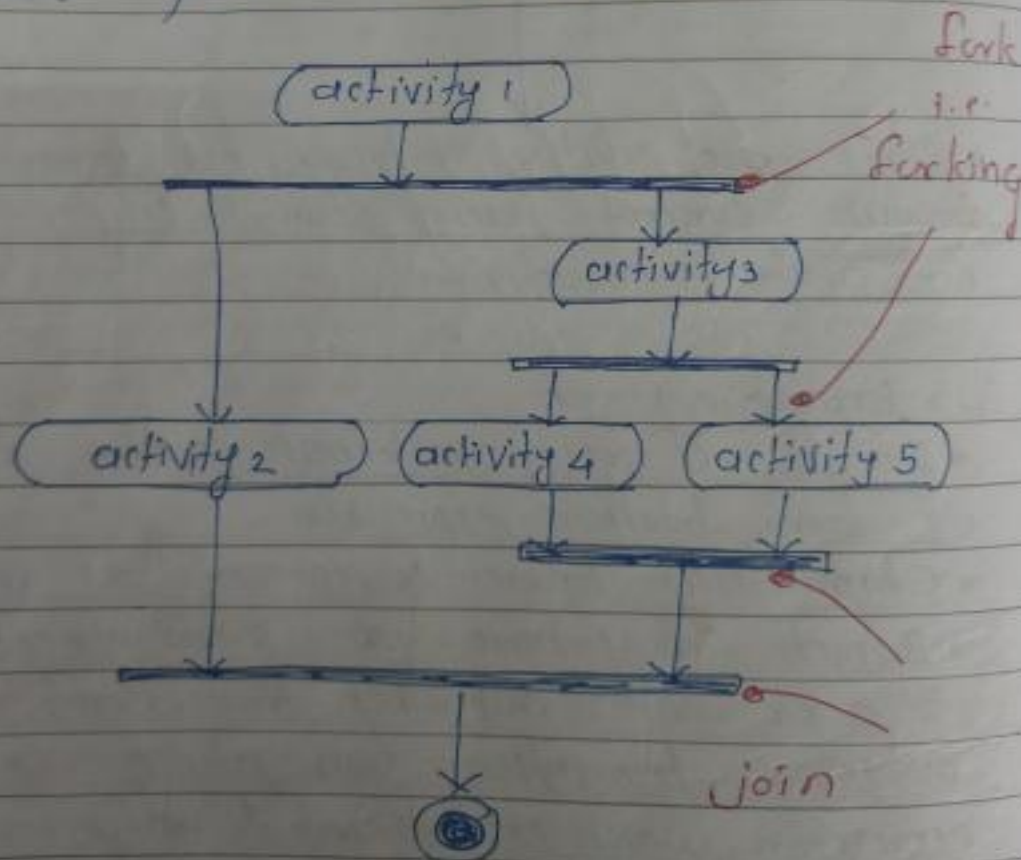
III) branching



IV) Forking and Joining

iv) Forking & Joining →

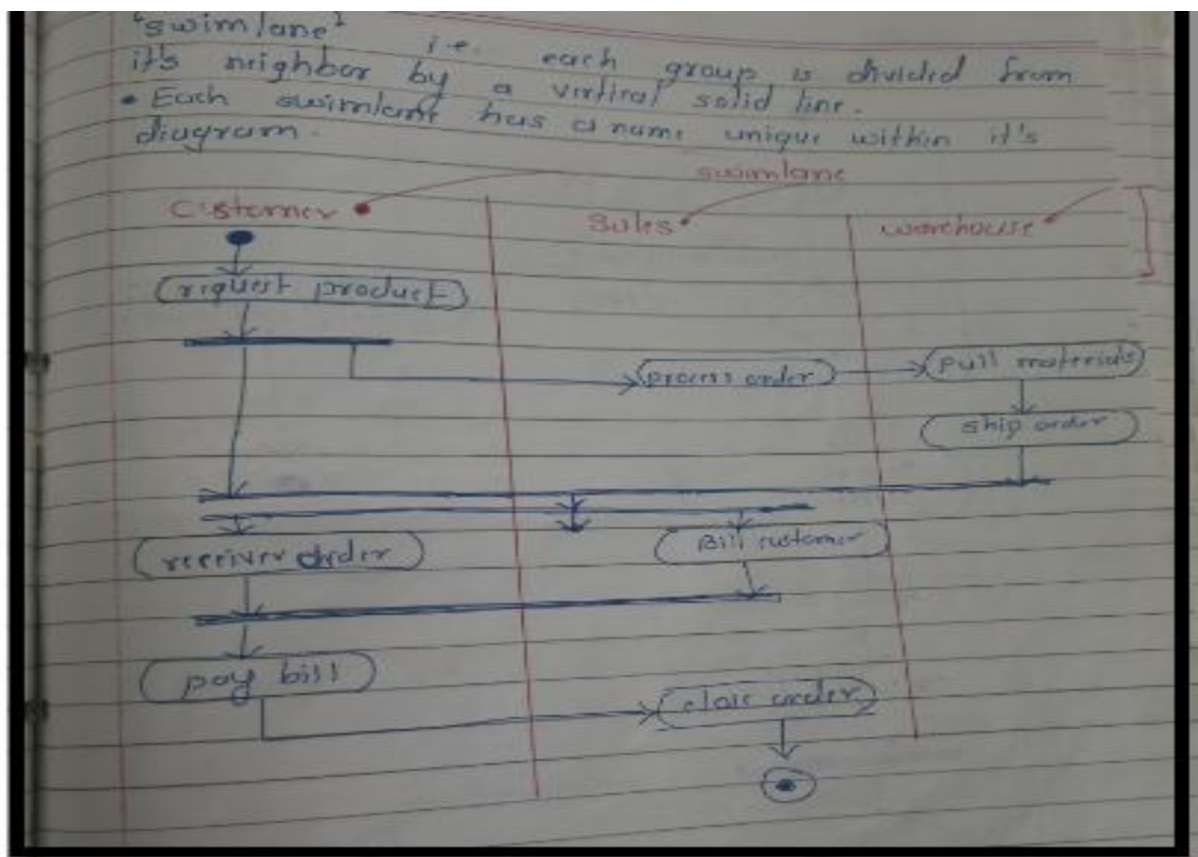
- When modelling work flow of business process — we might encounter flows that are concurrent.
- In UML, synchronization bar is used to specify the forking & joining of these parallel flow of control.



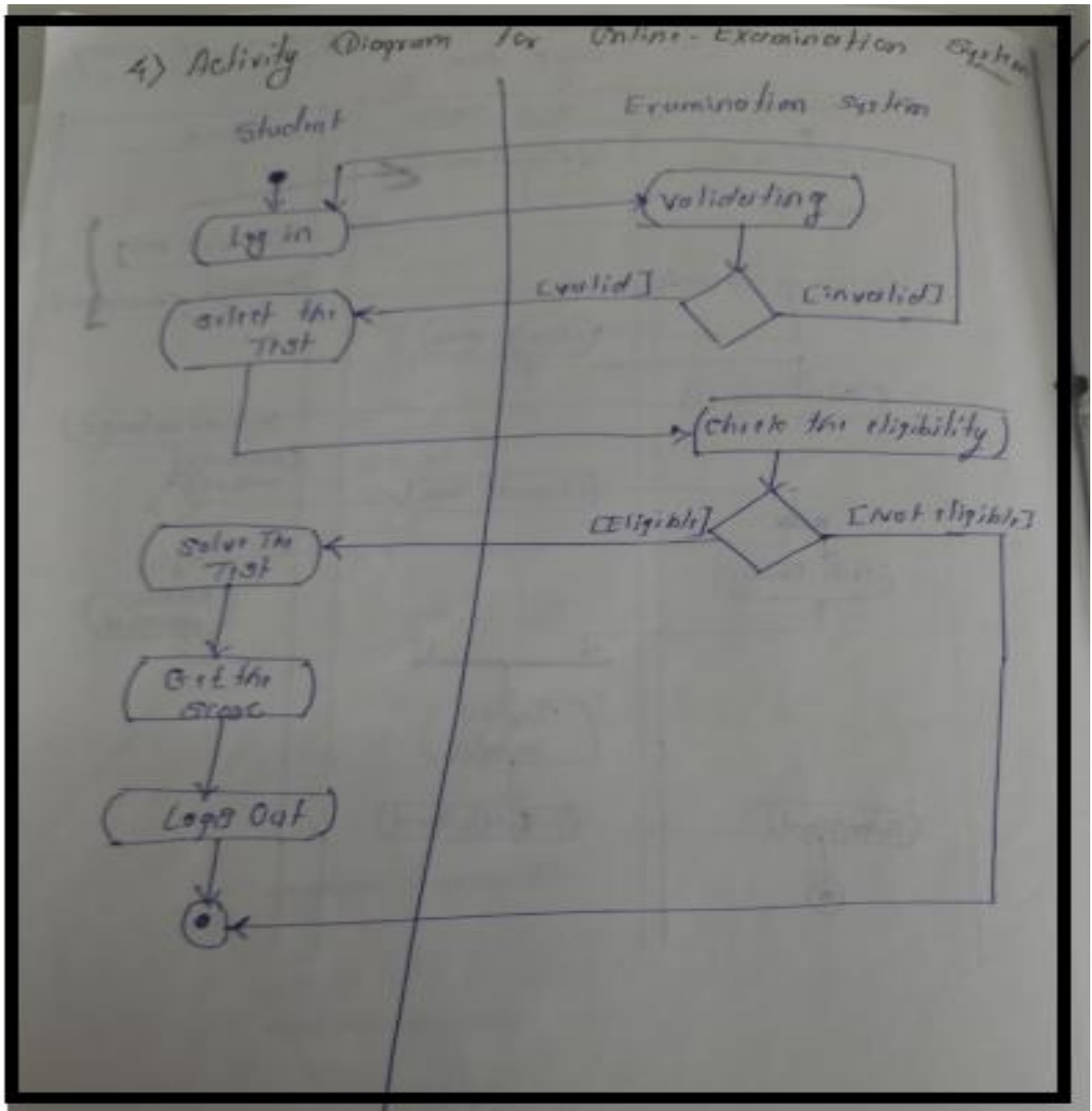
v) Swim-lane

V) Swimlanes → workflows of business processes, to partition the activity states on an activity diagrams divided into groups; each group representing the business organization responsible for those activities.

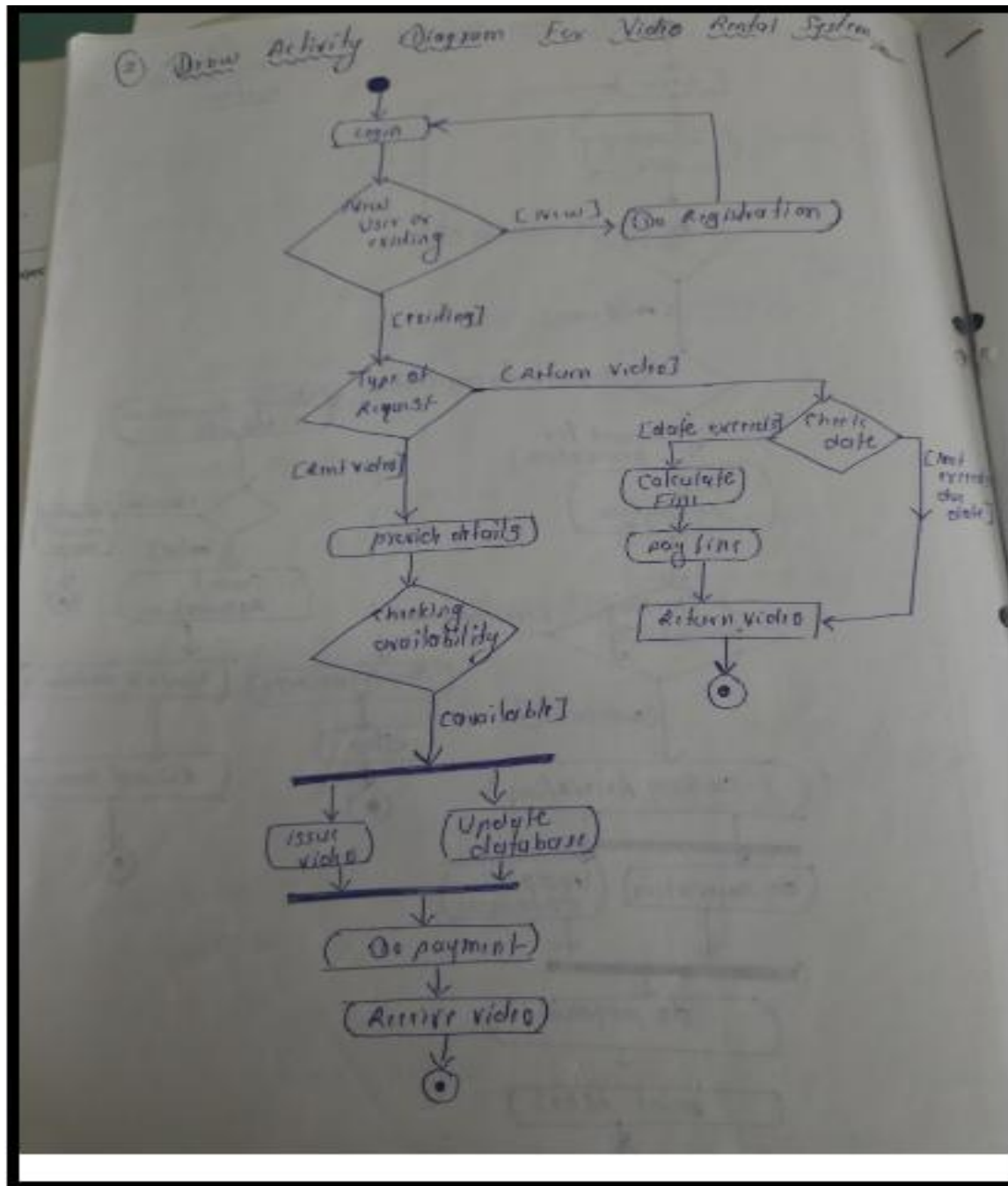
- In UML, each group is called swimlane.



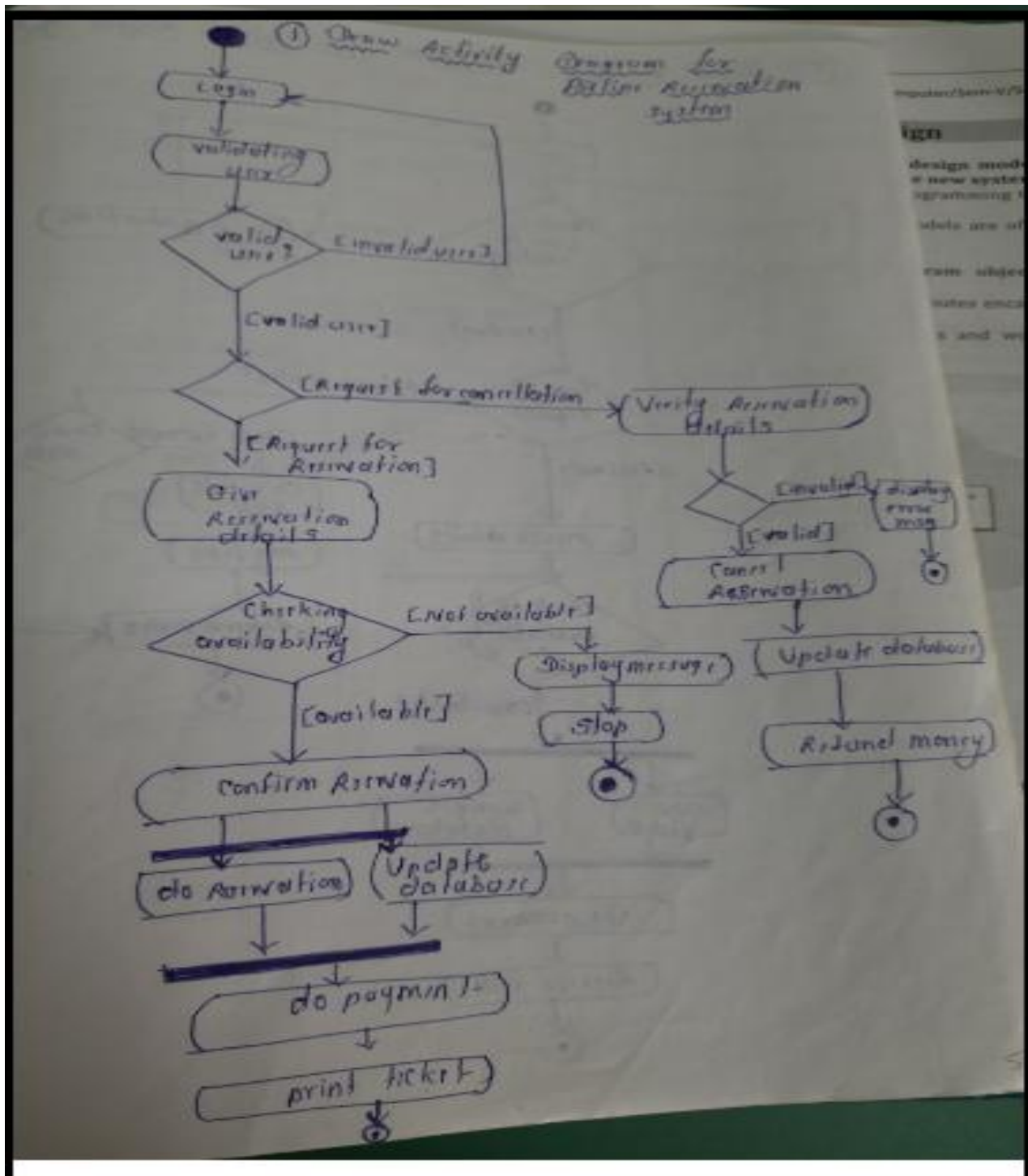
Example: Activity diagram for online examination system



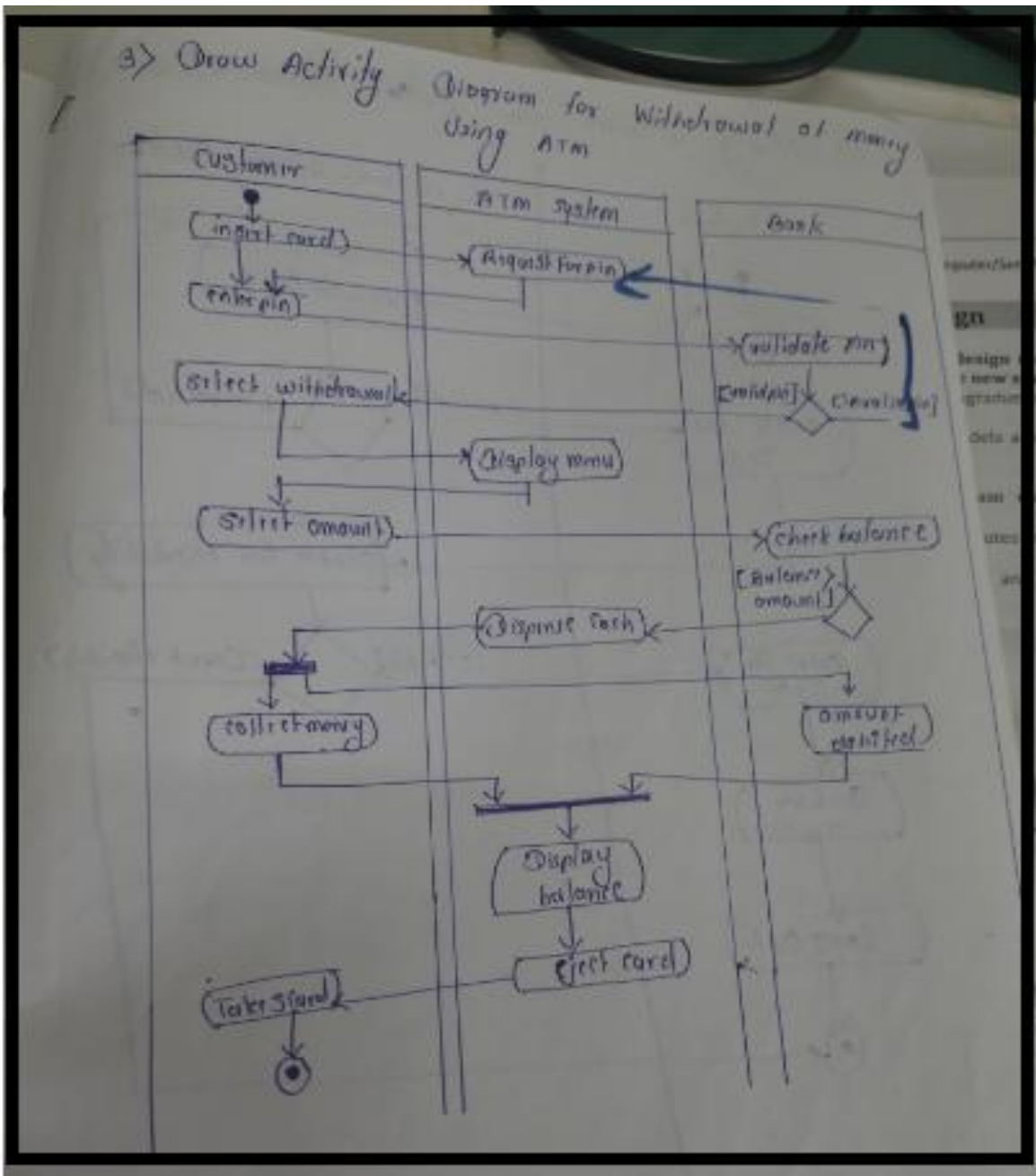
Example: Activity diagram for Video Rental system



Example: Activity diagram for Airline Reservation System

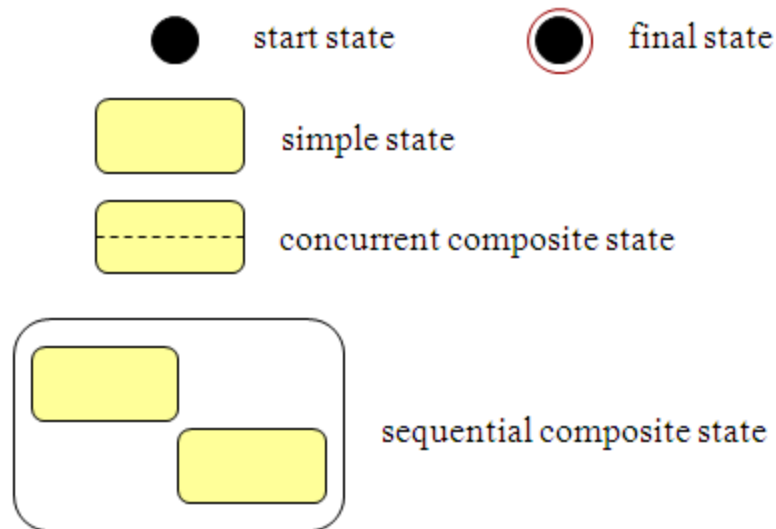


Example: Activity Diagram for Withdrawal of money using ATM

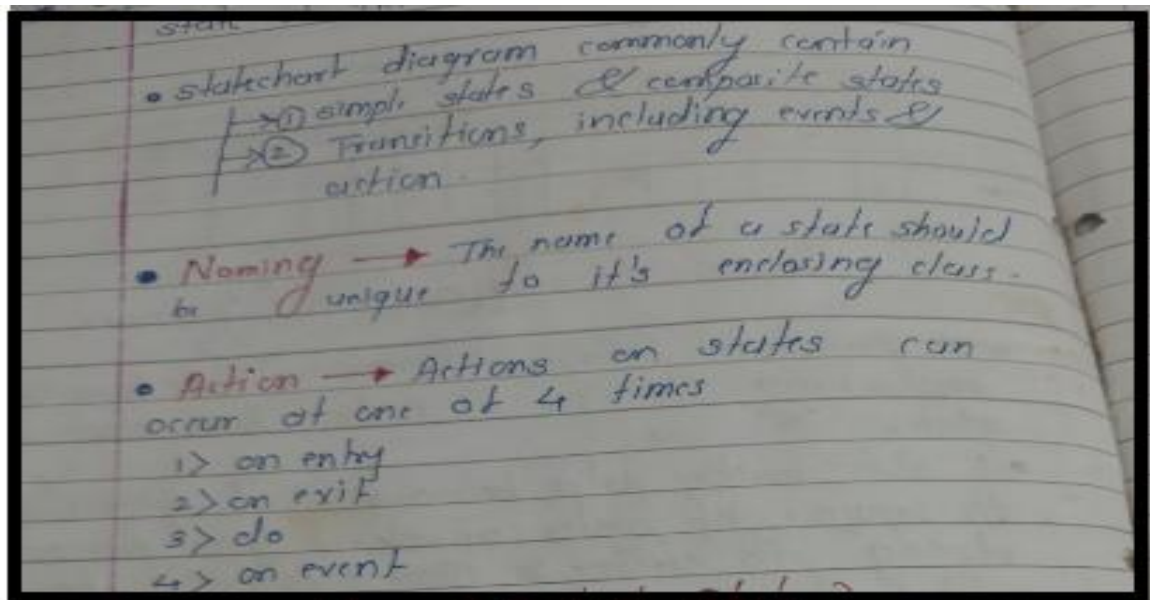
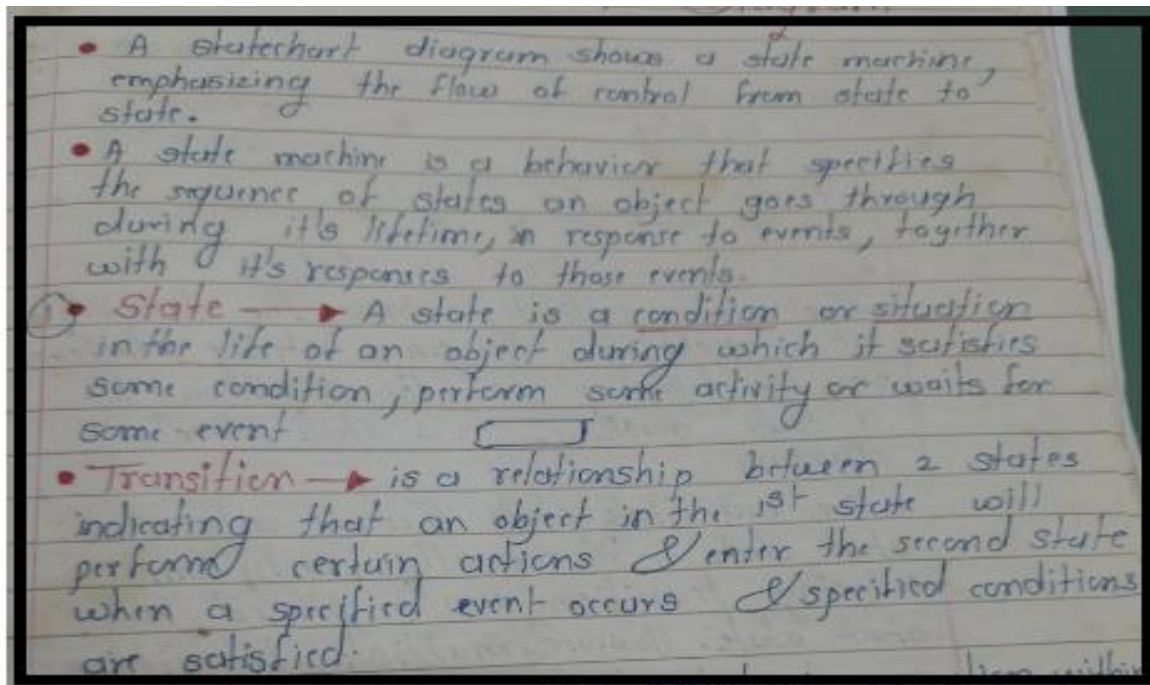


■ State Chart Diagram

- State Diagrams show the sequences of states an object goes through during its life cycle in response to stimuli, together with its responses and actions; an abstraction of all possible behaviors.
- State chart diagram consist of;
 - ✓ **Start State**
 - ✓ **Final/end state**
 - ✓ **Simple state/Composite state**
 - ✓ **Transition**



- **Introduction:**



- **Start state and End state**

• start state → (Initial State)

- start state explicitly shows the beginning of a workflow on an activity diagram or the beginning of the execution of a state machine on a state chart diagram

- For each state there is only 1 start state & normally, only one outgoing transition can be placed from start state. However, multiple transitions may be placed on start state if at least one of them is labeled with a condition. No incoming transitions

are allowed.

- graphically, start state is represented by filled circle

fig:- start state

- End state → • End state represents a final or terminal state on an activity or state chart diagram.

- place an end state when you want to explicitly show the end of workflow on an activity diagram or state chart diagram.

- There can be any no. of end states per context.

- Graphically it is represented by filled circle inside a slightly large unfilled circle that may contain a name (end process)

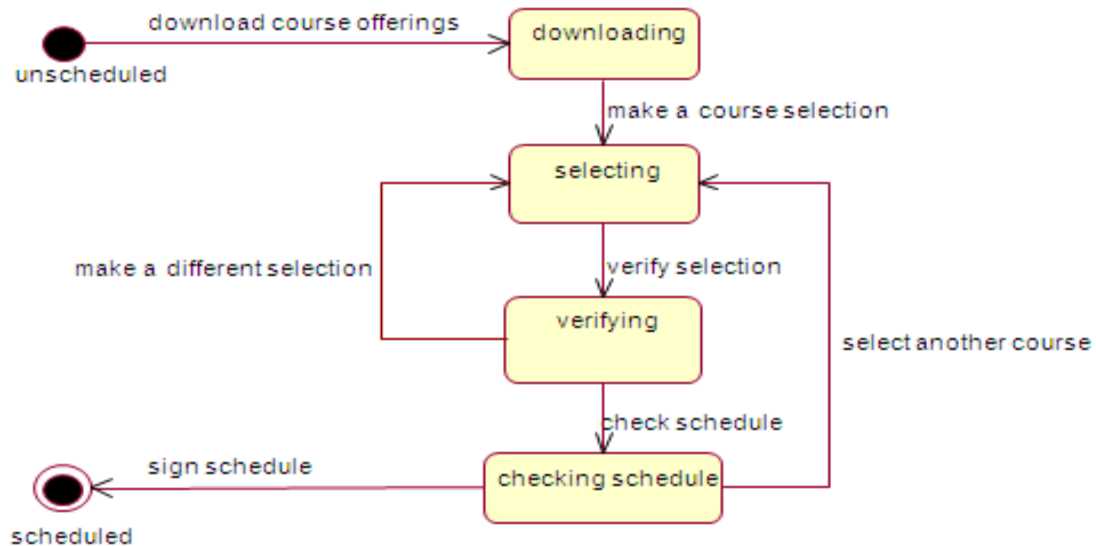


fig:- end state

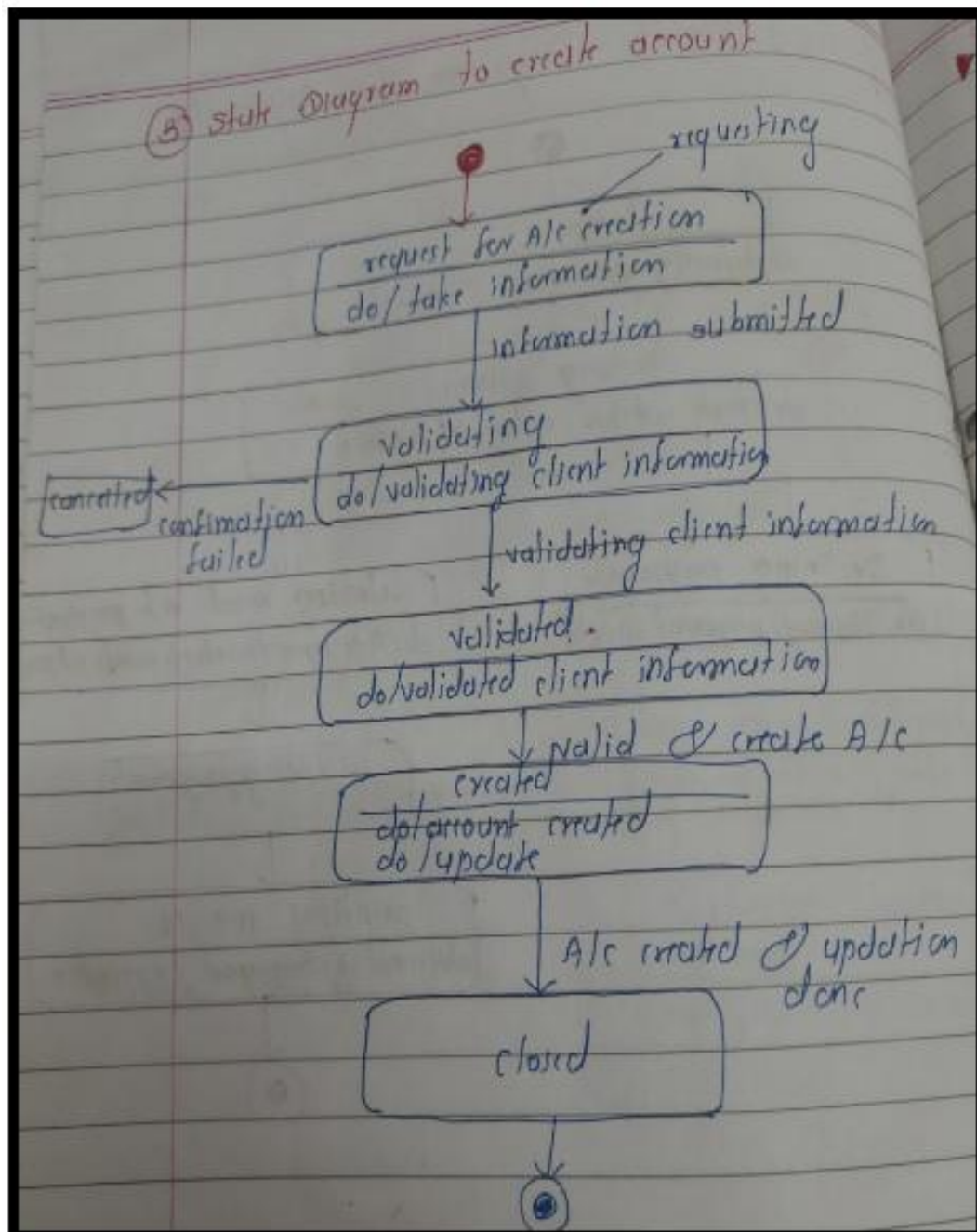
• Nested States

● Nested states → states may be nested to any depth level. Enclosing states are referred to as super states & everything that lies within the bounds of the super state is referred to as its contents. Nested states are called substates.

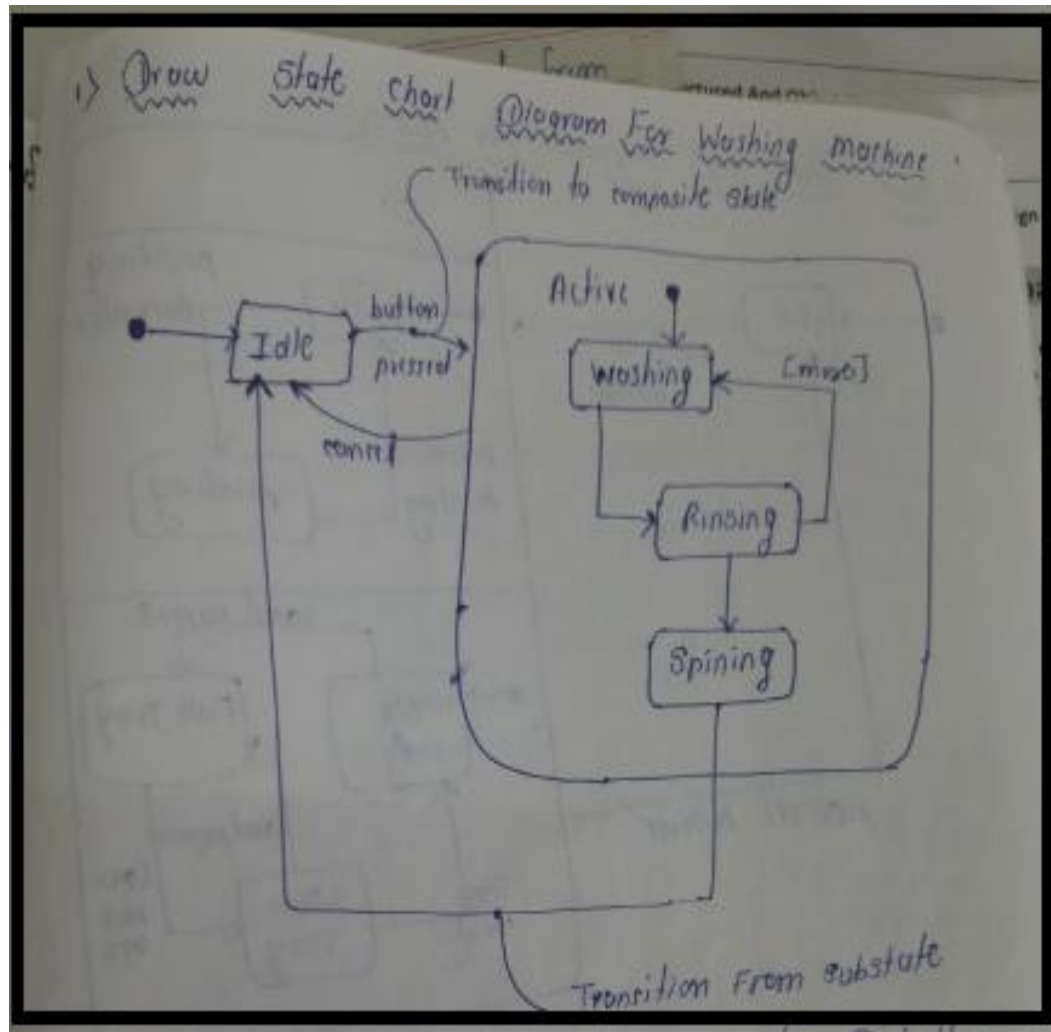
• Example



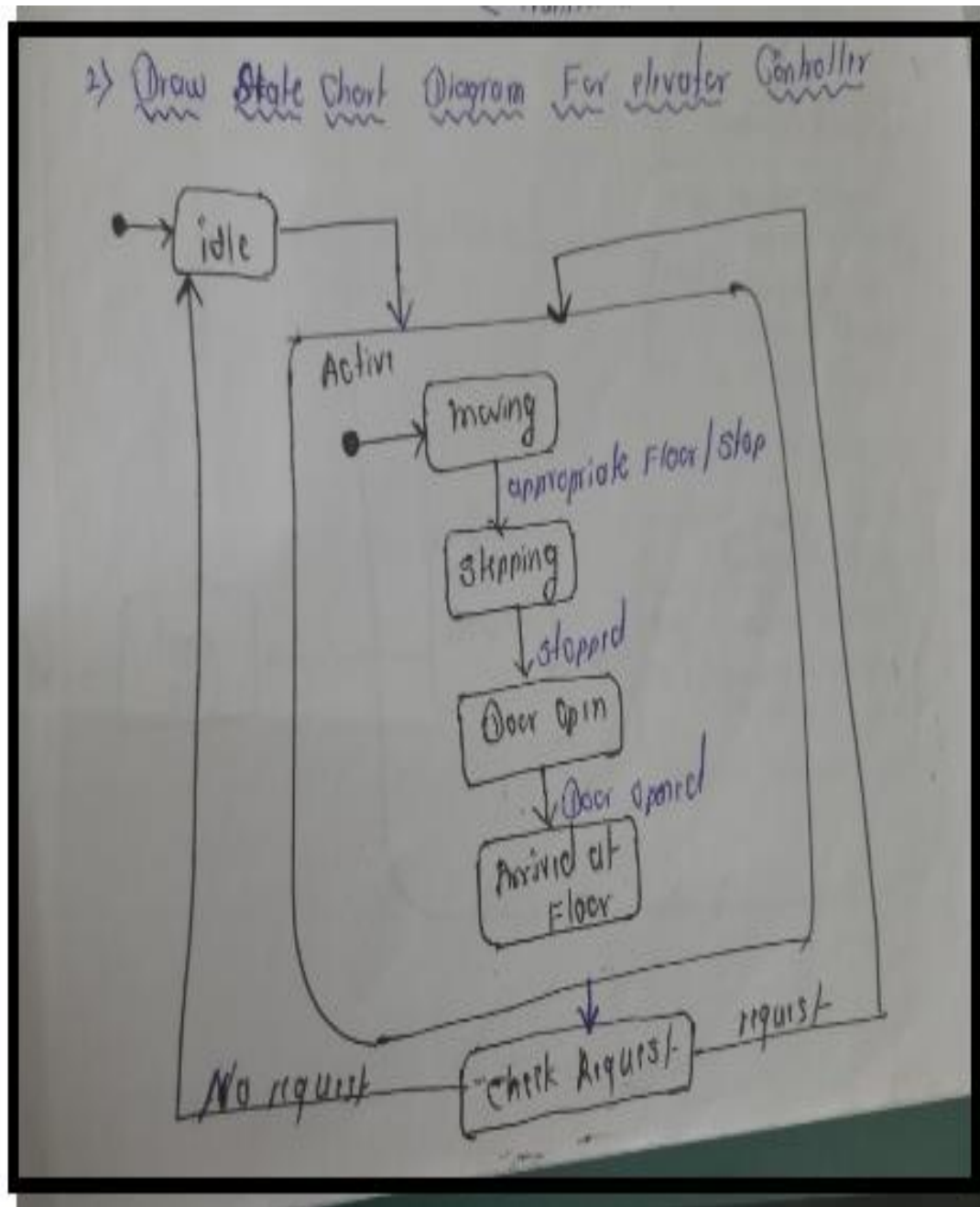
• Example:



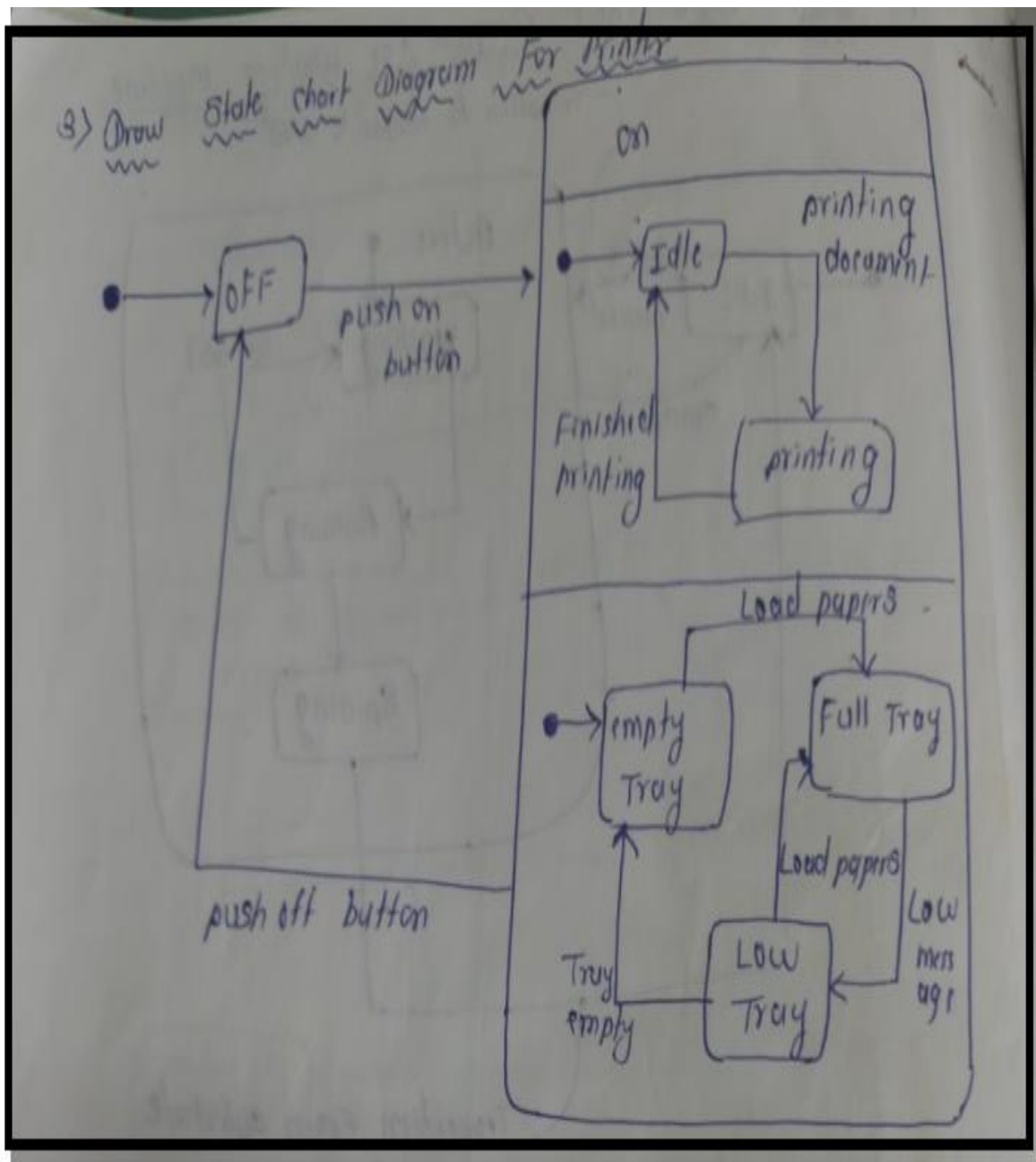
- **Example:**



- **Example:**



- Example:



■ UML Component Diagram

UML Component diagram describes the different elements required for implementing a system. Component is a modular unit with well-defined interfaces that is replaceable within its environment.

A component diagram shows a set of components and their relationships.

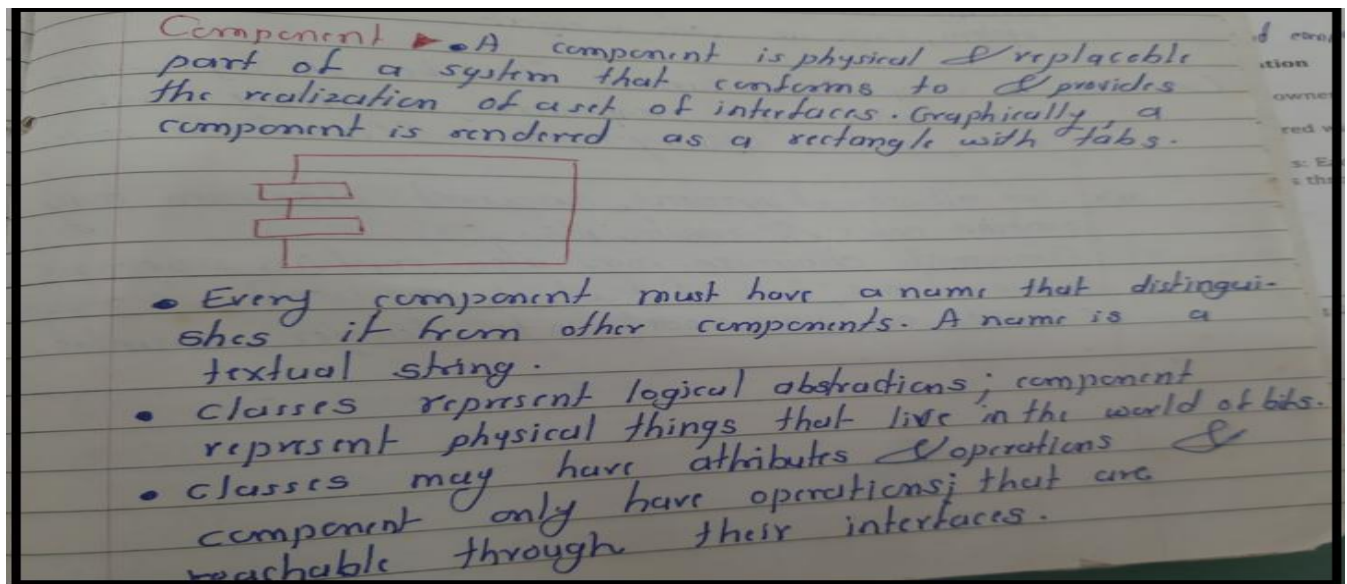
Graphically, a components diagram is a collection of vertices and arcs.

A components diagram is just a special kind of diagram and shares the same common properties as do all other diagrams-a name and graphical contents that are a projection into a model. What distinguishes a component diagram from all other kinds of diagrams is its particular content.

Component diagram commonly contain

- ✓ **Components**
- ✓ **Interfaces**
- ✓ **Dependency, generalization, association and realization relationships**

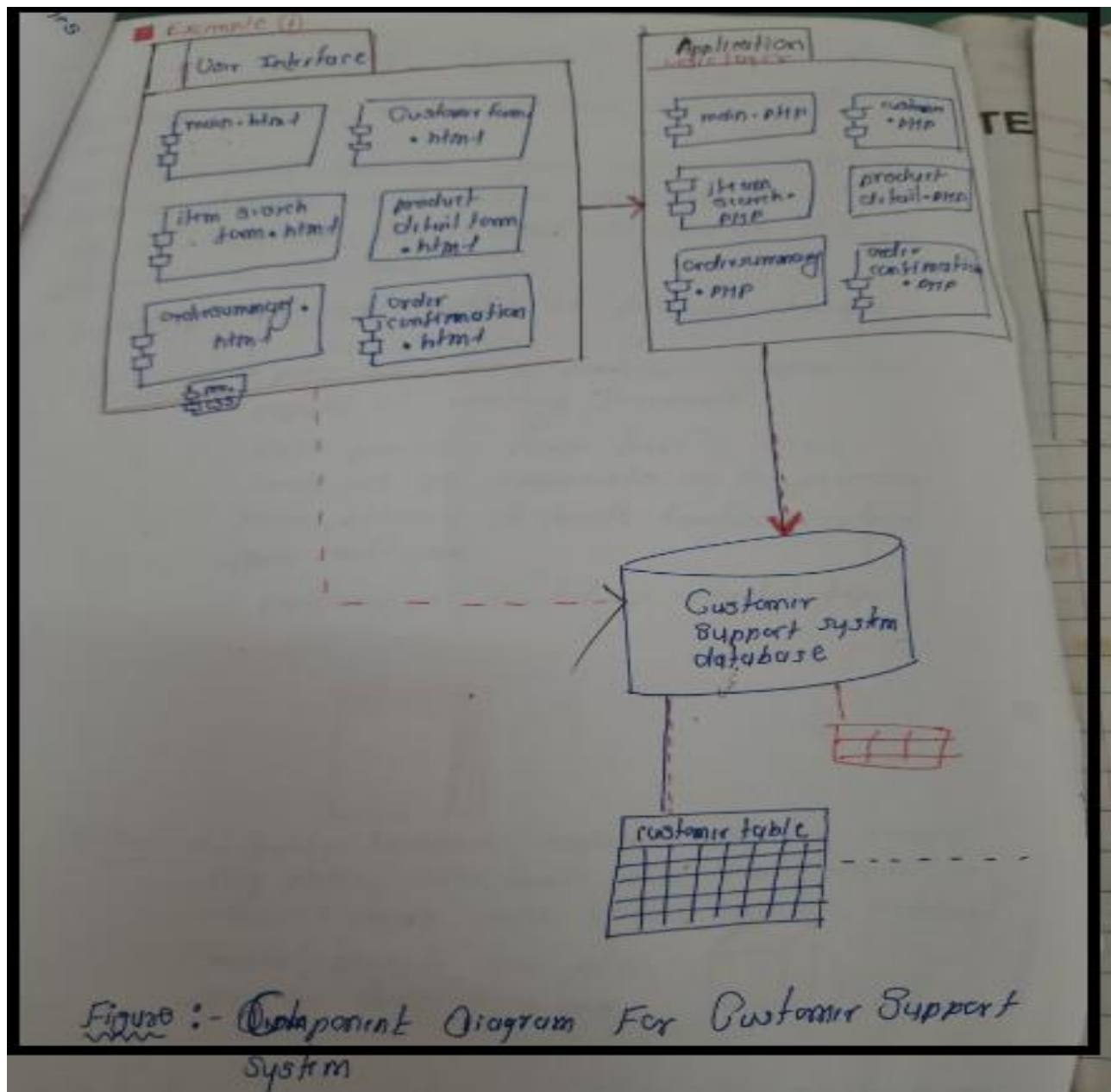
Component



▶ Diff. Types of Components

- ① Deployment components → These are components necessary & sufficient to form an executable system, such as dynamic libraries & executable.
- ② Work Product Components → These components are essentially the residue of the development process, consisting of things such as source code files & data files from which deployment-component are created.
- ③ Execution Components → These components are created has a consequences of an executing system, such as com + object.

Conclusion → Component diagram describe the organization of physical software components, including source code, runtime (binary) code & executable code.



■ UML Deployment Diagram

- A deployment diagram shows **processors, devices, and connections**. Each model contains a single deployment diagram which shows the connections between its processors and devices, and the allocation of its processes to processors.
- Processor Specifications, Device Specifications, and Connection Specifications enable you to display and modify the respective properties. The information in a specification is presented textually; some of this information can also be displayed inside the icons.
- You can change properties or relationships by editing the specification or modifying the icon on the diagram. The deployment diagram specifications are automatically updated.
- Deployment diagram are used to show the nodes where software components reside in the run-time system.
- **Deployment diagram consist of**

Nodes (Node may be Processor or Device)

+ Processor

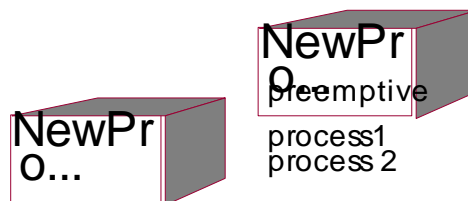
A processor is a hardware component capable of executing programs.

Naming

Each processor must have a name. There are no constraints on the processor name because processors denote hardware rather than software entities.

Graphical Depiction

The icon for a processor is a shaded box:



+ Device

A device is a hardware component with no computing power. Each device must have a name. Device names can be generic, such as "modem" or "terminal."

Graphical Depiction



Connection

✚ A connection depicts the communication path used by the hardware to communicate.

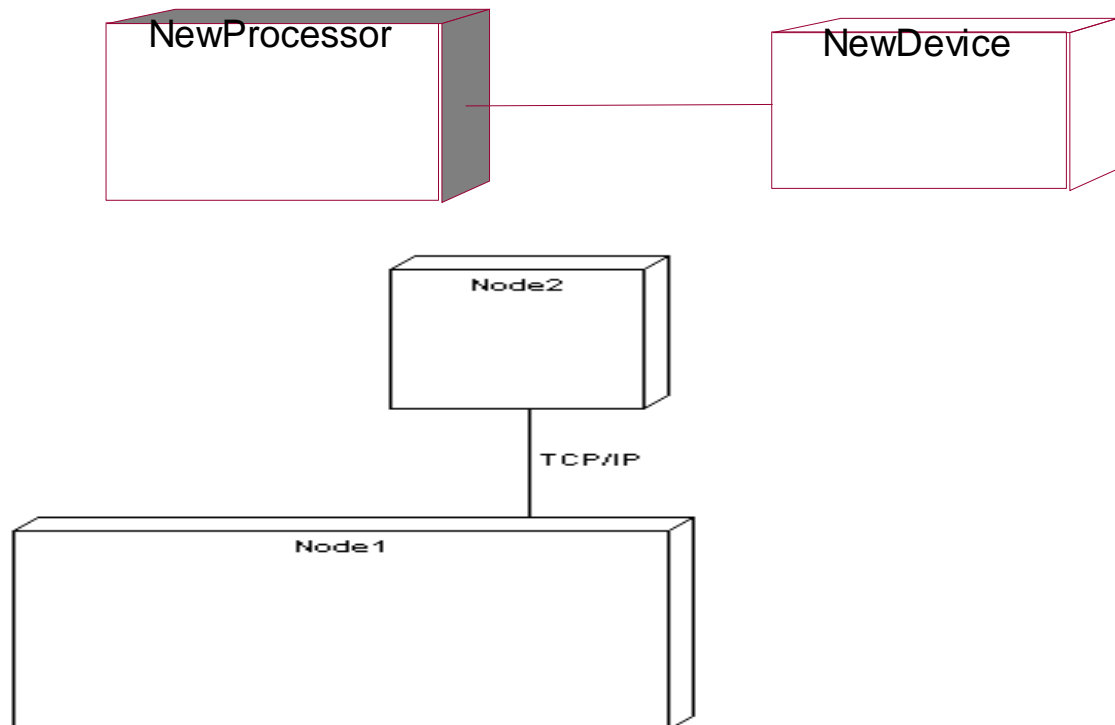
A connection represents some type of hardware coupling between two entities. An entity is either a processor or a device. The hardware coupling can be direct, such as an RS232 cable, or indirect, such as satellite-to-ground communication. Connections are usually bi-directional.

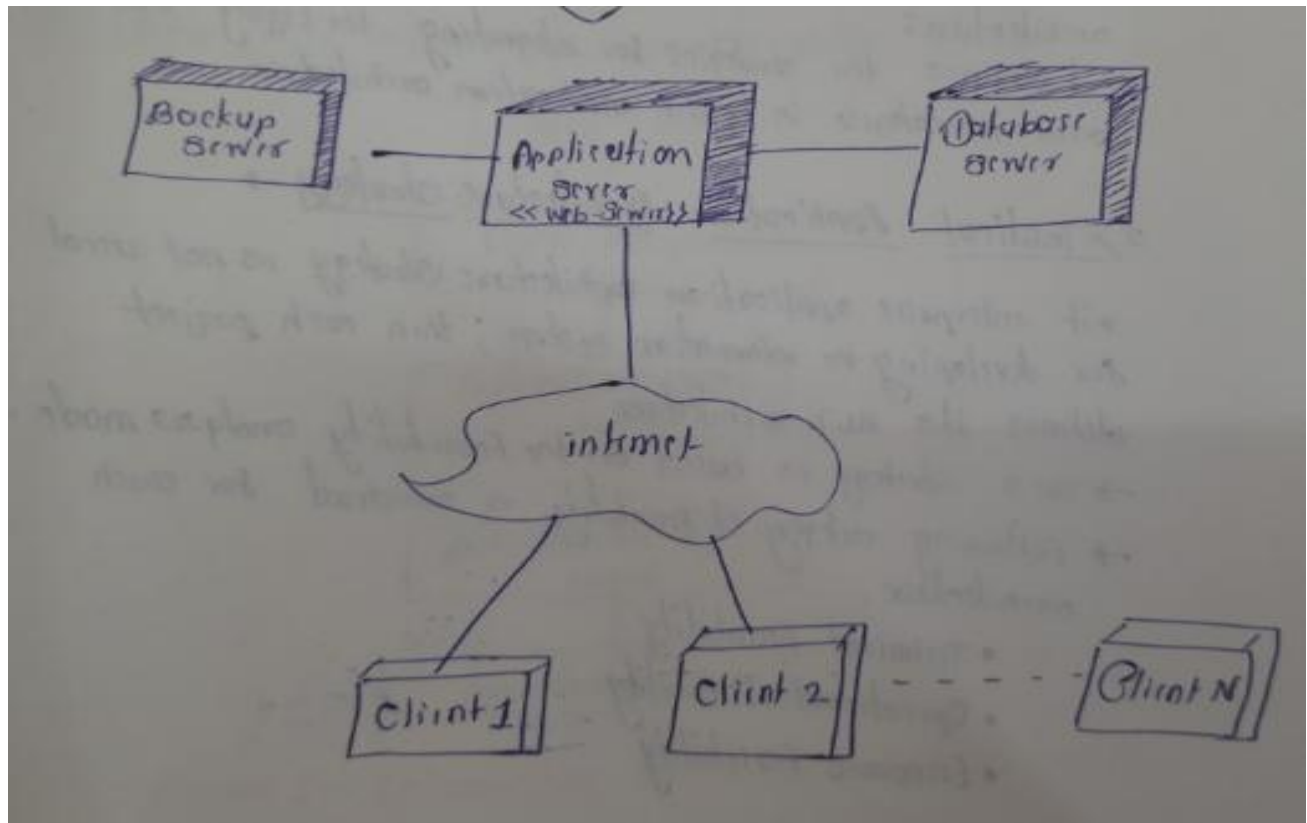
Naming`

You can optionally label the connection with its name.

Graphical Depiction

The icon for a connection is a straight line:





→ Deployment diagram consist of following elements,

- i) processor → processor is a hardware components capable of executing programs.
- Each processor must have a name.
 - There are no constraints on the processor name, because it denote hardware rather than software.
 - Processor is denoted by shaded box.
- e.g.



- ii) Device → Device is hardware component with no computing power.
- Each Device must have Name.
 - Device's name can be generic such as, "terminal", "printer", "scanner", "LAN", etc.
 - Icon for Device is a box.



- iii) Connection → Connection represent some type of hardware coupling between 2 entities.
- An entity is either processor or device.
 - Icon for a connection is a straight line.

