



VIT[®]
BHOPAL

Embedded Systems Modeling

Instructor: Dr. Ankur Beohar
School of Electrical and Electronics Engineering
VIT Bhopal University

Embedded Systems

■ Suggested Textbooks:

- ❑ Raj Kamal, “Embedded systems Architecture, Programming and Design”, Third Edition, Tata McGraw Hill, 2017.
- ❑ Rob Toulson and Tim Wilmshurst, “Fast and Effective Embedded Systems Design – Applying the ARM mbed”, Elsevier, 2017.
- ❑ Arnold S. Berger, “Embedded Systems Design: An Introduction to Processes, Tools, and Techniques”, CRC Press, 2002.

■ Other sources

- ❑ Lecture notes
- ❑ Handouts
- ❑ Blogs
- ❑ MOOC courses



Finite State Machine Model

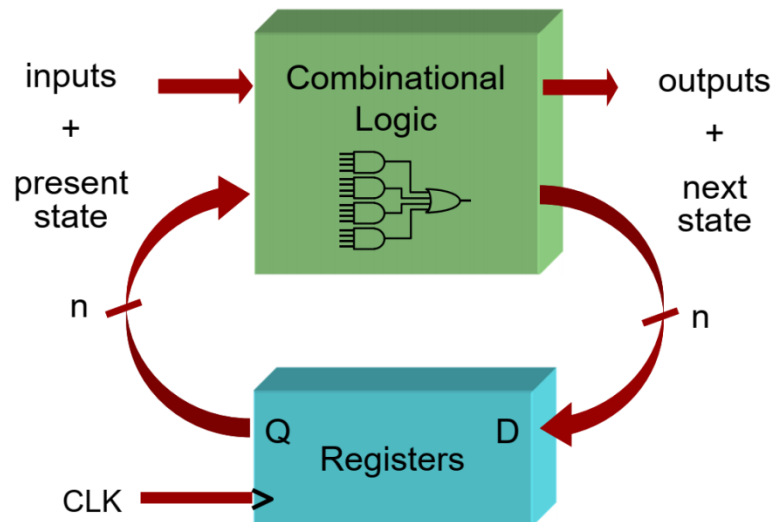


Finite State Machine

- A finite-state machine (FSM) or finite-state automaton (FSA, plural: automata), finite automaton, is a mathematical model of computation.
- It is an abstract machine that can be in exactly one of a finite number of states at any given time.
- The FSM can change from one state to another in response to some inputs; the change from one state to another is called a transition.
- An FSM is defined by a list of its states, its initial state, and the inputs that trigger each transition.
- Two types—deterministic finite-state machines and non-deterministic finite-state machines.

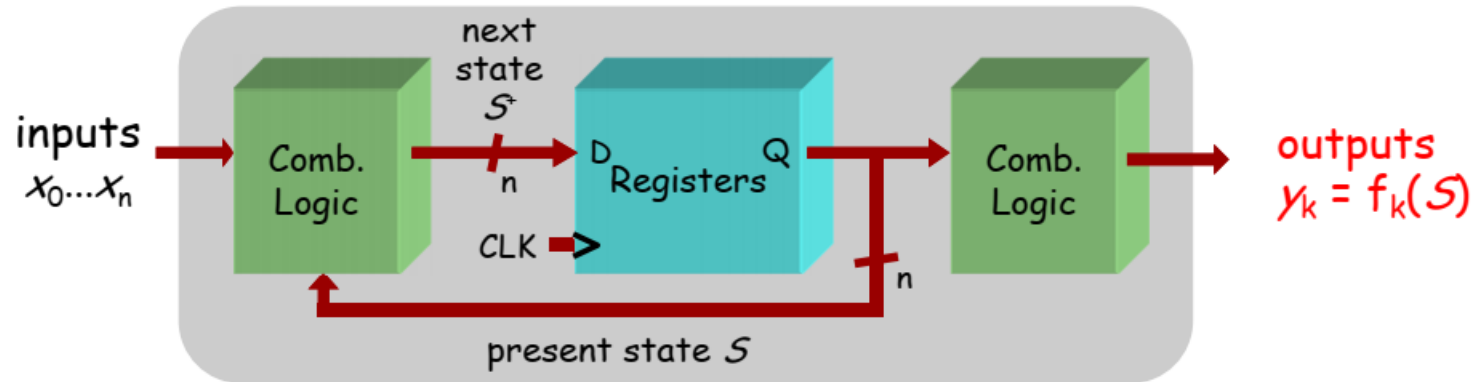
Finite State Machine

- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized “states” of operation
- At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state

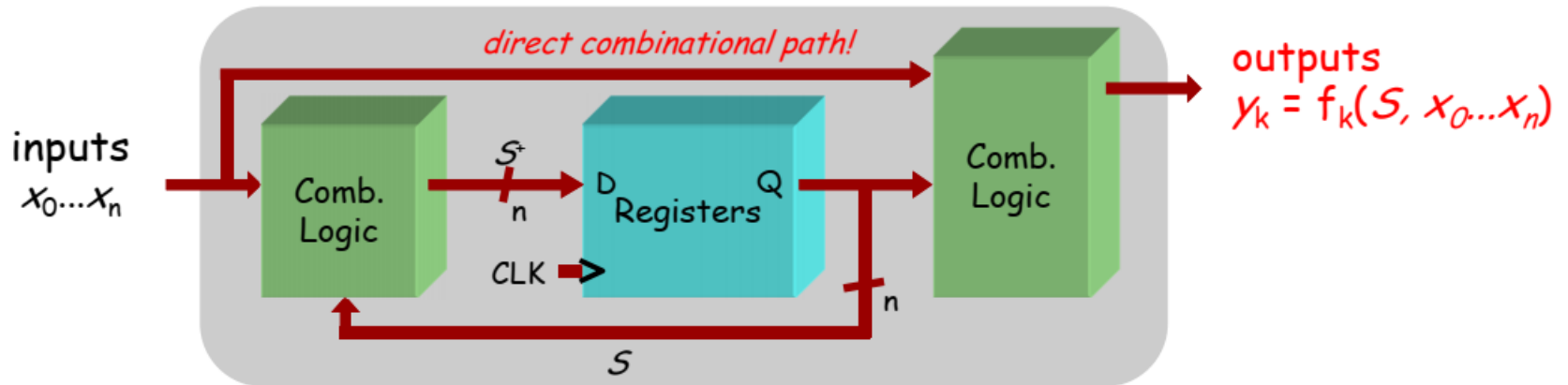


Two Types

- Moore FSM:

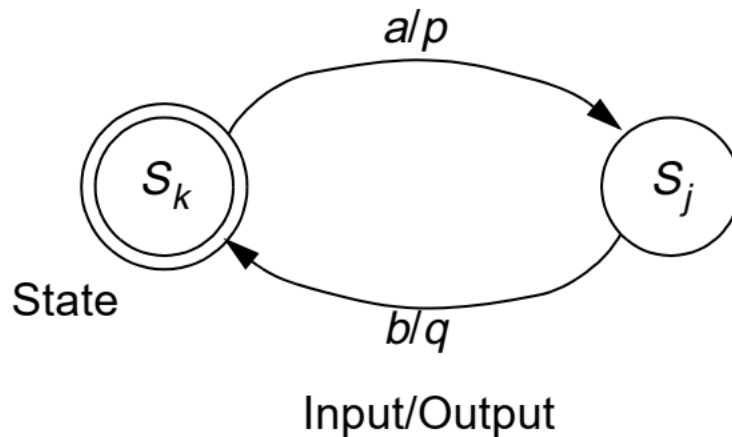


- Mealy FSM:



State Diagrams

- A state diagram represents a finite state machine (FSM) and contains
 - Circles: represent the machine states
 - Labelled with a binary encoded number or S_k reflecting state.
 - Directed arcs: represent the transitions between states
 - Labelled with input/output for that state transition.



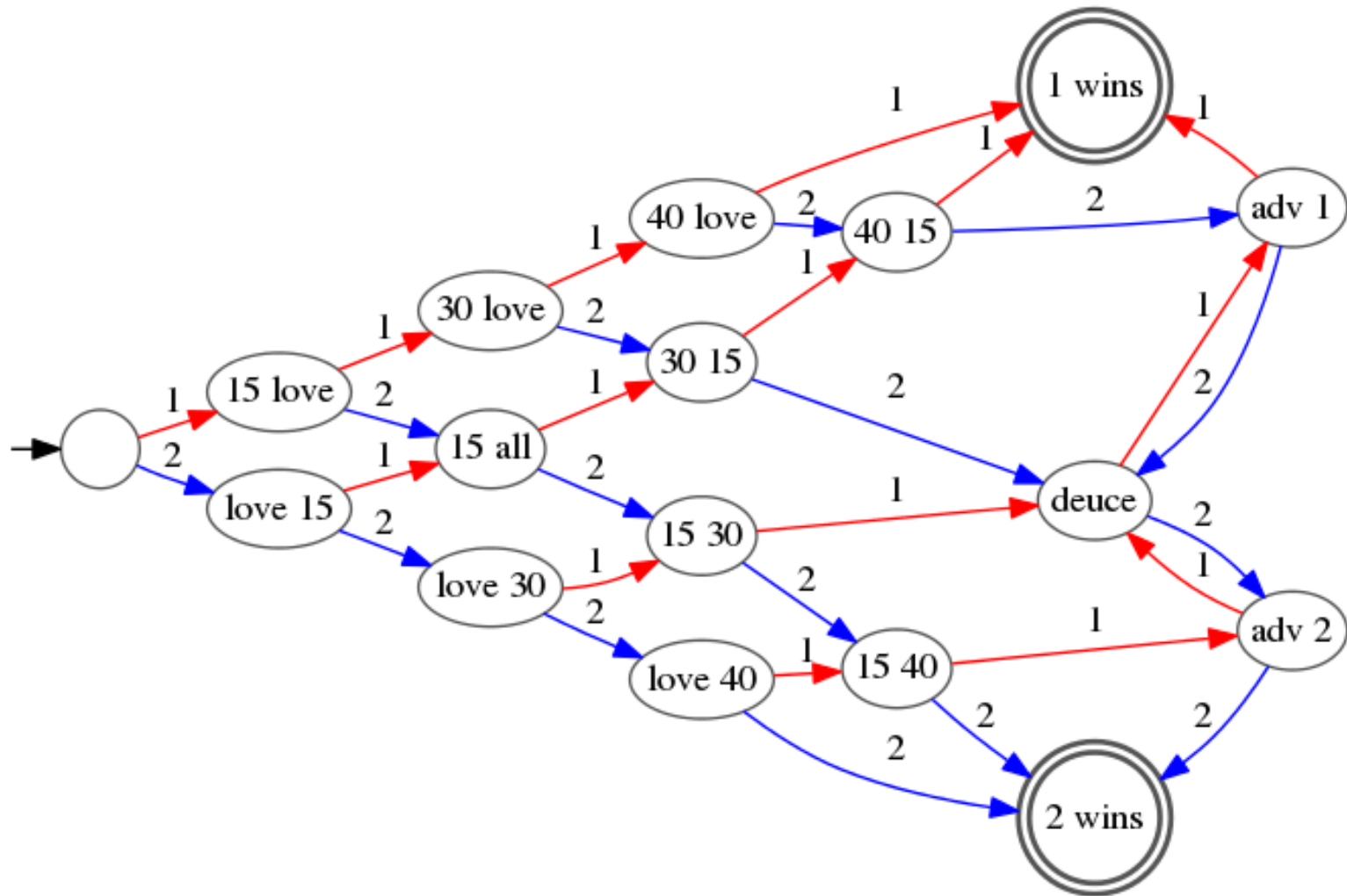
Input: $x(t) \in \{a, b\}$
Output: $z(t) \in \{p, q\}$
State: $s(t) \in \{S_k, S_j\}$
Initial state: $s(0) = S_k$

Tennis Game

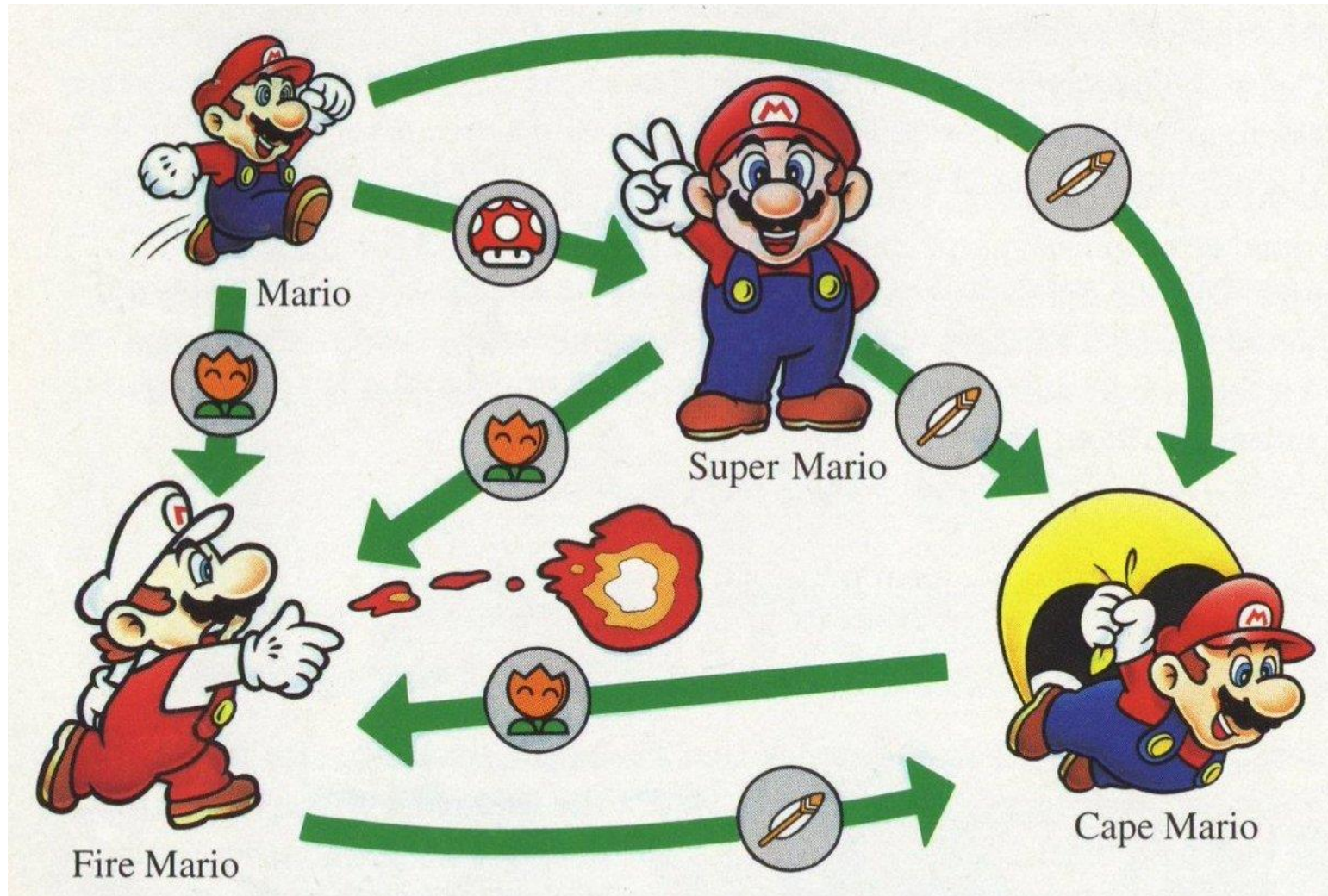
- The game is easy to explain: to win, you must score at least 4 points and win by at least 2. Yet in practice, you are supposed to use strange labels like "love" (0 points), "15" (1 point), "30" (2 points), "40" (3 points), "deuce" (3 or more points each, and the players are tied), "all" (players are tied) instead of simply tracking points as numbers, as other sports do.

The diagram illustrates a game tree for a two-player game. The root node is a white circle. Player 1 (red arrows) moves first, choosing between 1 and 2. If Player 1 chooses 1, Player 2 (blue arrows) moves, choosing between 'love' and 'all'. If Player 1 chooses 2, Player 2 chooses between 'love' and '30'. The game continues with alternating moves until a terminal state is reached. Terminal states are '1 wins' and '2 wins'. There are also nodes labeled 'adv 1', 'adv 2', and 'deuce'.

Simplified One!!



Mario Game



Mario Game

- **States**

- Mario (We will refer as Small Mario hereafter)
- Super Mario
- Fire Mario
- Cape Mario
- Lost Life (Apart from the image considering this state)

- **Events**

- Got Mushroom
- Got Fire Flower
- Got Feather
- Met Monster (Not shown in image)

In Continuation...

Current State	Event Occurred	New State	Coins Earned
Small Mario	Got Mushroom	Super Mario	100
Small Mario	Got Fire Flower	Fire Mario	200
Small Mario	Got Feather	Cape Mario	300
Small Mario	Met Monster	Lost Life	0
Super Mario	Got Mushroom	Super Mario	100
Super Mario	Got Fire Flower	Fire Mario	200
Super Mario	Got Feather	Cape Mario	300
Super Mario	Met Monster	Small Mario	0
Fire Mario	Got Mushroom	Fire Mario	100
Fire Mario	Got Fire Flower	Fire Mario	200
Fire Mario	Got Feather	Cape Mario	300
Fire Mario	Met Monster	Small Mario	0
Cape Mario	Got Mushroom	Cape Mario	100
Cape Mario	Got Fire Flower	Fire Mario	200
Cape Mario	Got Feather	Cape Mario	300
Cape Mario	Met Monster	Small Mario	0

```
■ public class Mario {
■     enum internalState {
■         SmallMario,
■         SuperMario,
■         FireMario,
■         CapeMario
■     }
■
■     public int LifeCount { get; private set; }
■     public int CoinCount { get; private set; }
■     private internalState State { get; set; }
■
■     public Mario() {
■         LifeCount = 1;
■         CoinCount = 0;
■         State = internalState.SmallMario;
■     }
■
■     public void GotMushroom() {
■         WriteLine("Got Mushroom!");
■         if (State == internalState.SmallMario)
■             State = internalState.SuperMario;
■
■         GotCoins(100);
■     }
■ }
```

- **public void** GotFireFlower() {
- WriteLine("Got FireFlower!");
- State = internalState.FireMario;
- GotCoins(200);
- }
-
- **public void** GotFeather() {
- WriteLine("Got Feather!");
- State = internalState.CapeMario;
- GotCoins(300);
- }
-
- **public void** GotCoins(int numberOfCoins) {
- WriteLine(\$"Got {numberOfCoins} Coin(s)!");
- CoinCount += numberOfCoins;
- **if** (CoinCount >= 5000)
- {
- GotLife();
- CoinCount -= 5000;
- }
- }
-
-

```
■ private void GotLife() {
■     WriteLine("Got Life!");
■     LifeCount += 1;
■ }
■
■ private void LostLife() {
■     WriteLine("Lost Life!");
■     LifeCount -= 1;
■     if (LifeCount <= 0)
■         GameOver();
■ }
■
■ public void MetMonster() {
■     WriteLine("Met Monster!");
■     if (State == internalState.SmallMario)
■         LostLife();
■     else
■         State = internalState.SmallMario;
■ }
■
■ public void GameOver() {
■     LifeCount = 0;
■     CoinCount = 0;
■     WriteLine("Game Over!");
■ }
■
■ public override string ToString() {
■     return $"State: {State} | LifeCount: {LifeCount} | CoinsCount: {CoinCount} \n";
```



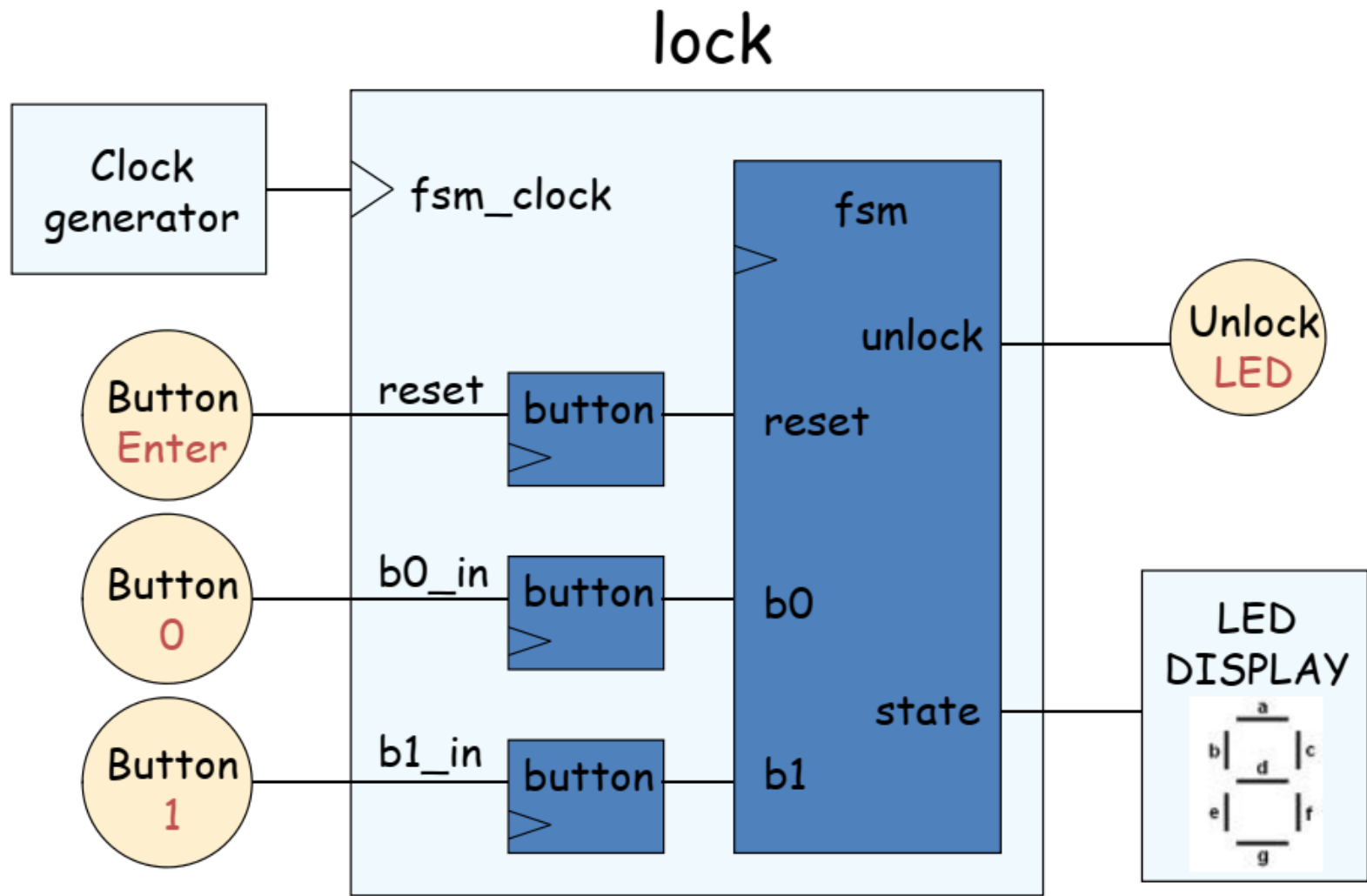
```
■ class MainClass {  
■   static void Main(string[] args) {  
■       Mario mario = new Mario();  
■       WriteLine(mario);  
■  
■       mario.GotMushroom();  
■       WriteLine(mario);  
■  
■       mario.GotFireFlower();  
■       WriteLine(mario);  
■  
■       mario.GotFeather();  
■       WriteLine(mario);  
■  
■       mario.GotCoins(4800);  
■       WriteLine(mario);  
■  
■       mario.MetMonster();  
■       WriteLine(mario);  
■  
■       mario.MetMonster();  
■       WriteLine(mario);
```

Example:

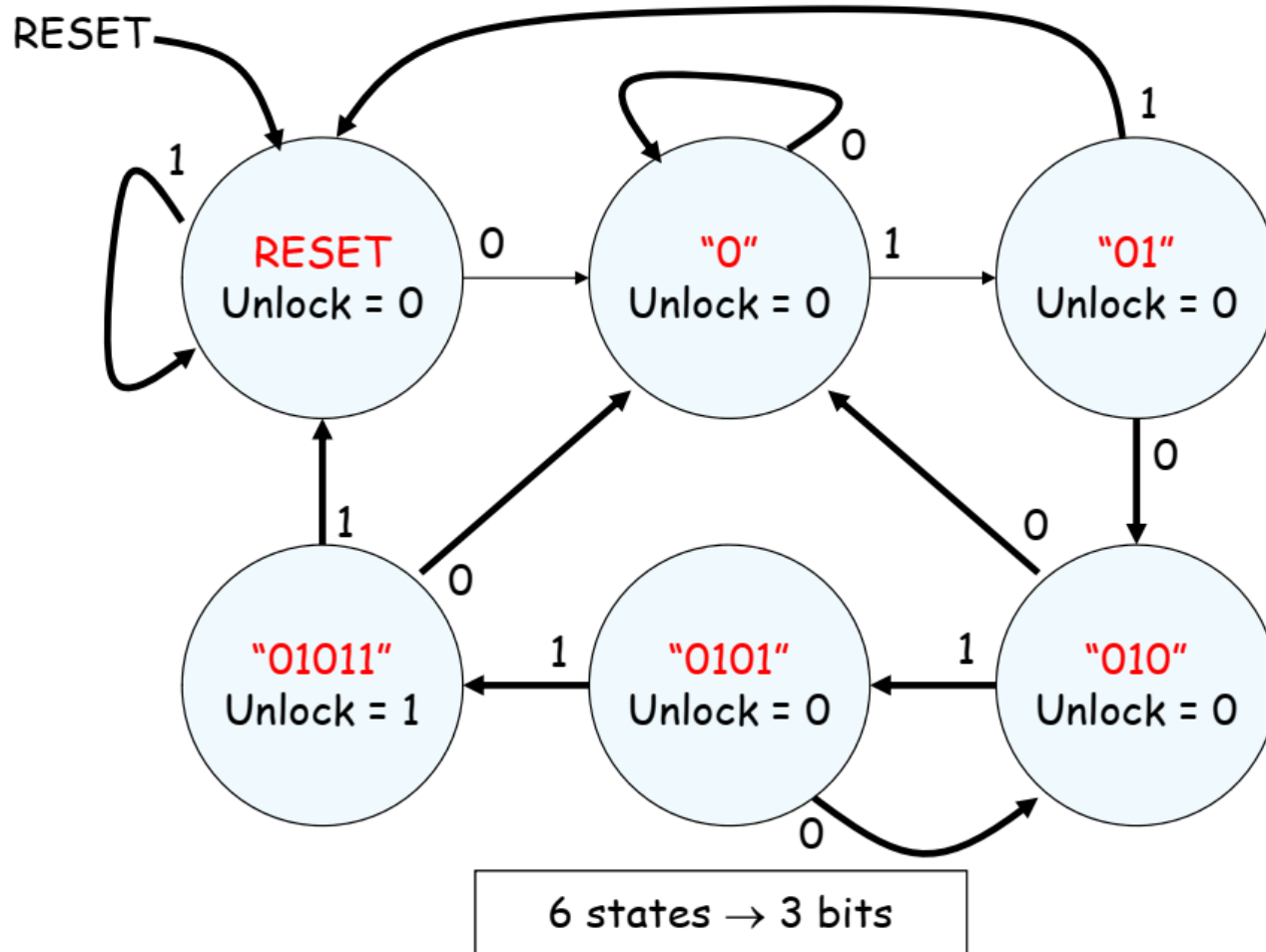
- Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination should be 01011.



Block diagram



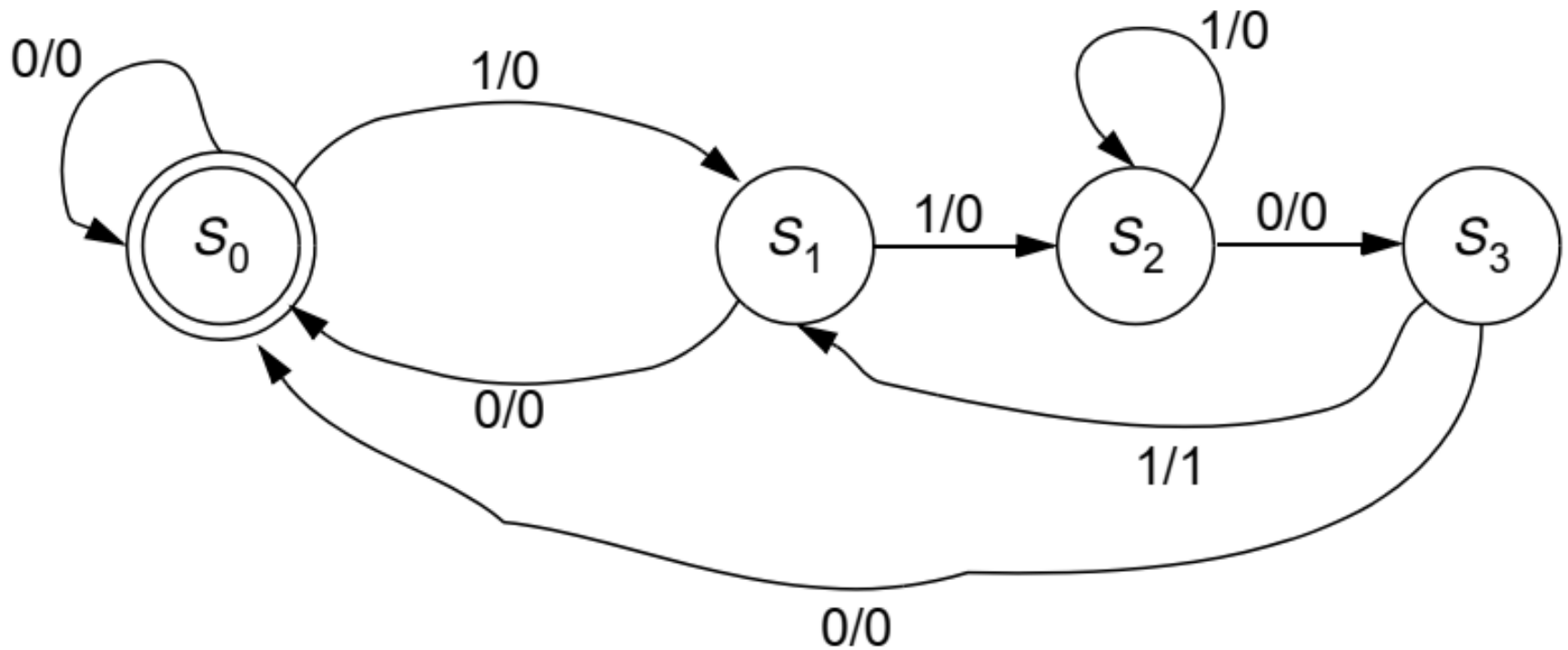
State Transition Diagram



Question

Design a sequential system which can detect 1101 sequence?

Solution



State Transition Table

Present State P₁ P₀			Input X	Next State N₁ N₀			Output Z
S_0 or	0	0	0	S_0 or	0	0	0
S_0 or	0	0	1	S_1 or	0	1	0
S_1 or	0	1	0	S_0 or	0	0	0
S_1 or	0	1	1	S_2 or	1	0	0
S_2 or	1	0	0	S_3 or	1	1	0
S_2 or	1	0	1	S_2 or	1	0	0
S_3 or	1	1	0	S_0 or	0	0	0
S_3 or	1	1	1	S_1 or	0	1	1

Designing

		P_1P_0			
		00	01	11	10
X	0	0	0	1	0
	1	1	1	0	0

N_1

		P_1P_0			
		00	01	11	10
X	0	0	1	0	0
	1	1	0	1	0

N_0

		P_1P_0			
		00	01	11	10
X	0	0	0	0	0
	1	0	0	1	0

Z

$$N_1 = X\bar{P}_1 + \bar{X}P_1P_0$$

$$N_0 = \bar{X}\bar{P}_1P_0 + XP_1P_0 + X\bar{P}_1\bar{P}_0 = \bar{X}\bar{P}_1P_0 + X(\bar{P}_1 \oplus P_0)$$

$$Z = XP_1P_0$$

Designing

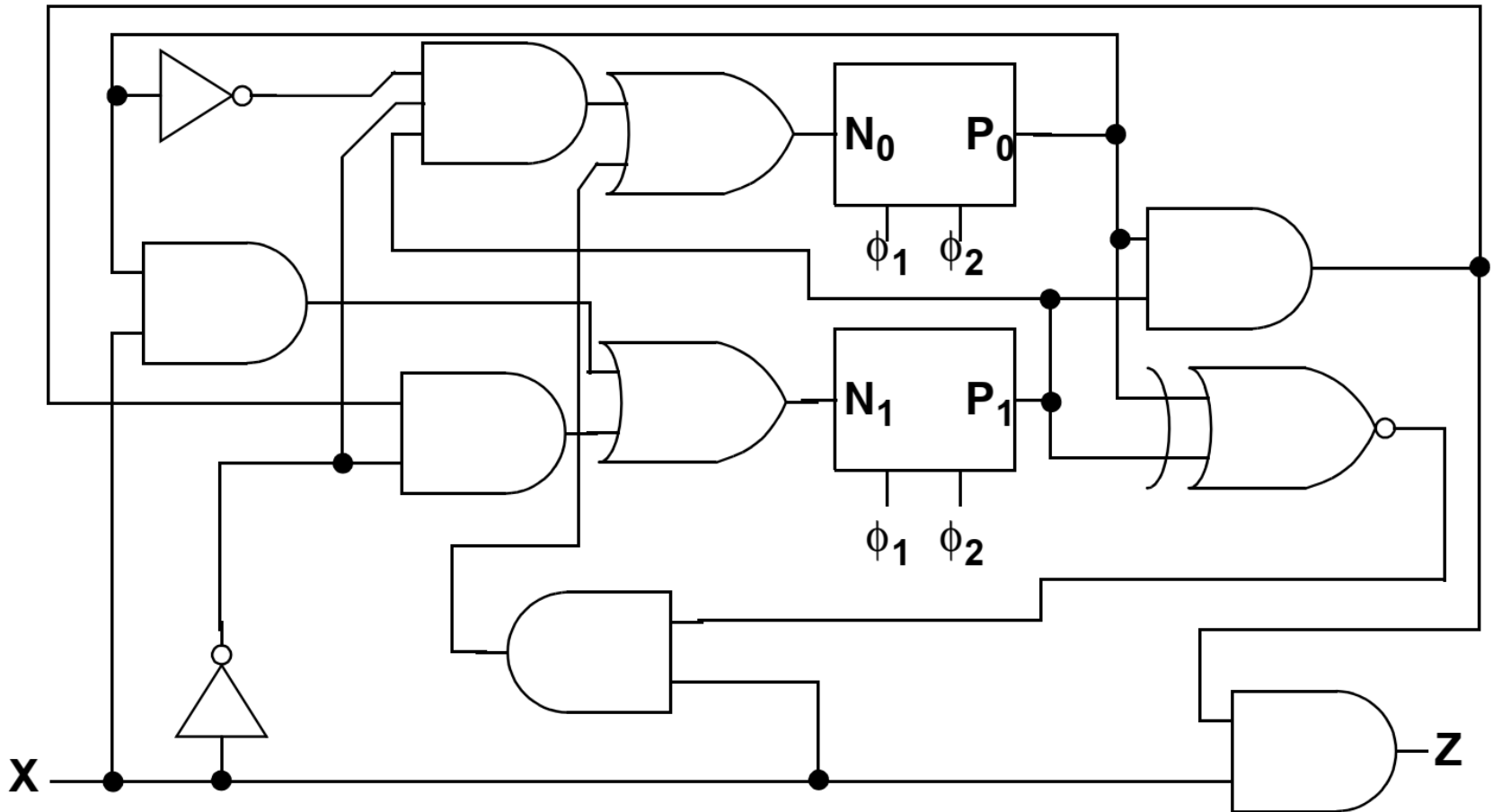
- Notice that the previous Boolean functions can also be expressed with time as follows.

$$\mathbf{N}_1(t) = \mathbf{P}_1(t+1) = \mathbf{X}(t) \cdot \overline{\mathbf{P}_1(t)} + \overline{\mathbf{X}(t)} \cdot \mathbf{P}_1(t) \cdot \mathbf{P}_0(t)$$

$$\begin{aligned}\mathbf{N}_0(t) = \mathbf{P}_0(t+1) &= \overline{\mathbf{X}(t)} \cdot \overline{\mathbf{P}_1(t)} \cdot \mathbf{P}_0(t) + \mathbf{X}(t) \cdot \mathbf{P}_1(t) \cdot \mathbf{P}_0(t) \\ &\quad + \mathbf{X}(t) \cdot \overline{\mathbf{P}_1(t)} \cdot \overline{\mathbf{P}_0(t)} \\ &= \overline{\mathbf{X}(t)} \cdot \overline{\mathbf{P}_1(t)} \cdot \mathbf{P}_0(t) + \mathbf{X}(t) \cdot \overline{\mathbf{P}_1(t)} \oplus \mathbf{P}_0(t)\end{aligned}$$

$$\mathbf{Z}(t) = \mathbf{X} \cdot \mathbf{P}_1(t) \cdot \mathbf{P}_0(t)$$

Designing



Question

Design a sequential system which can detect 10101 sequence?



Data Flow Graph Model



Data Flow Graph Model

- A Data Flow Diagram/ Graph (DFD) is a traditional visual representation of the information flows within a system.
- During the early stages of a project the systems analysis and design process gathers a great deal of unstructured data and references to processing requirements. It is important that all this should be summarised. The summary should serve to:-
 - Simplify communication with the end-user/customer.
 - Support the future development of the system.

Data Flow Graph Model

- By mapping out your process or system's flow of data, DFDs help you better understand your process or system, uncover its kinks, improve it, and can even help you implement a new process or system.
- It serves as an excellent communication tool that even non-technical users can understand. It is a good starting point for system design.

Data Flow Graph Model

- DFDs became popular in the 1970s and have been able to maintain their widespread use by being easy to understand. Visually displaying how a process or system works can hold people's attention and explain complex concepts better than blocks of text can, so DFDs are able to help almost anyone grasp a system or process' logic and functions.
- A neat and clear DFD can depict a good amount of the system requirements graphically.
- It can be manual, automated, or a combination of both.
- The DFG is also called as a data flow diagram or bubble chart.

DFG Symbols

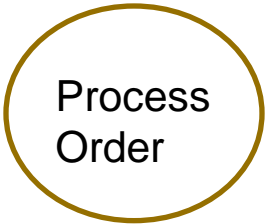
- A data flow diagram uses four basic symbols.
 - External entity
 - Process
 - Data store
 - Data flow

External entity

Customer

- An external entity can represent a human, system or application.
- It is where certain data comes from or goes to.
- They're either the sources or destinations of information. So they're usually placed on the diagram's edges.
- The sources from which information flows into the system and the recipients (or destination or sink) of information leaving the system.
- External entities are notated as rectangles.

Process



Process
Order



Process Order

- A process is an automated action or manual activity which takes some input and produces some output.
- A process is an activity or function where the manipulation and transformation of data takes place. A process can be decomposed to finer level of details, for representing how data is being processed within the process.
- Process is a procedure that manipulates the data and its flow by taking incoming data, changing it, and producing an output with it. A process can do this by performing computations and using logic to sort the data or change its flow of direction.
- Something happens to the data during a process.
- Notated as a rectangle with rounded corners or a circle/bubble.

Data Store

Inventory

Inventory

- Data stores show where required or produced data related to the process is stored.
- Data stores hold information for later use, like a file of documents that's waiting to be processed. Data inputs flow through a process and then through a data store while data outputs flow out of a data store and then through a process.
- Permanent or semi-permanent store for data/information – a file of something or a variable.
- Can be notated in any of the two ways as shown above.

Data Flow

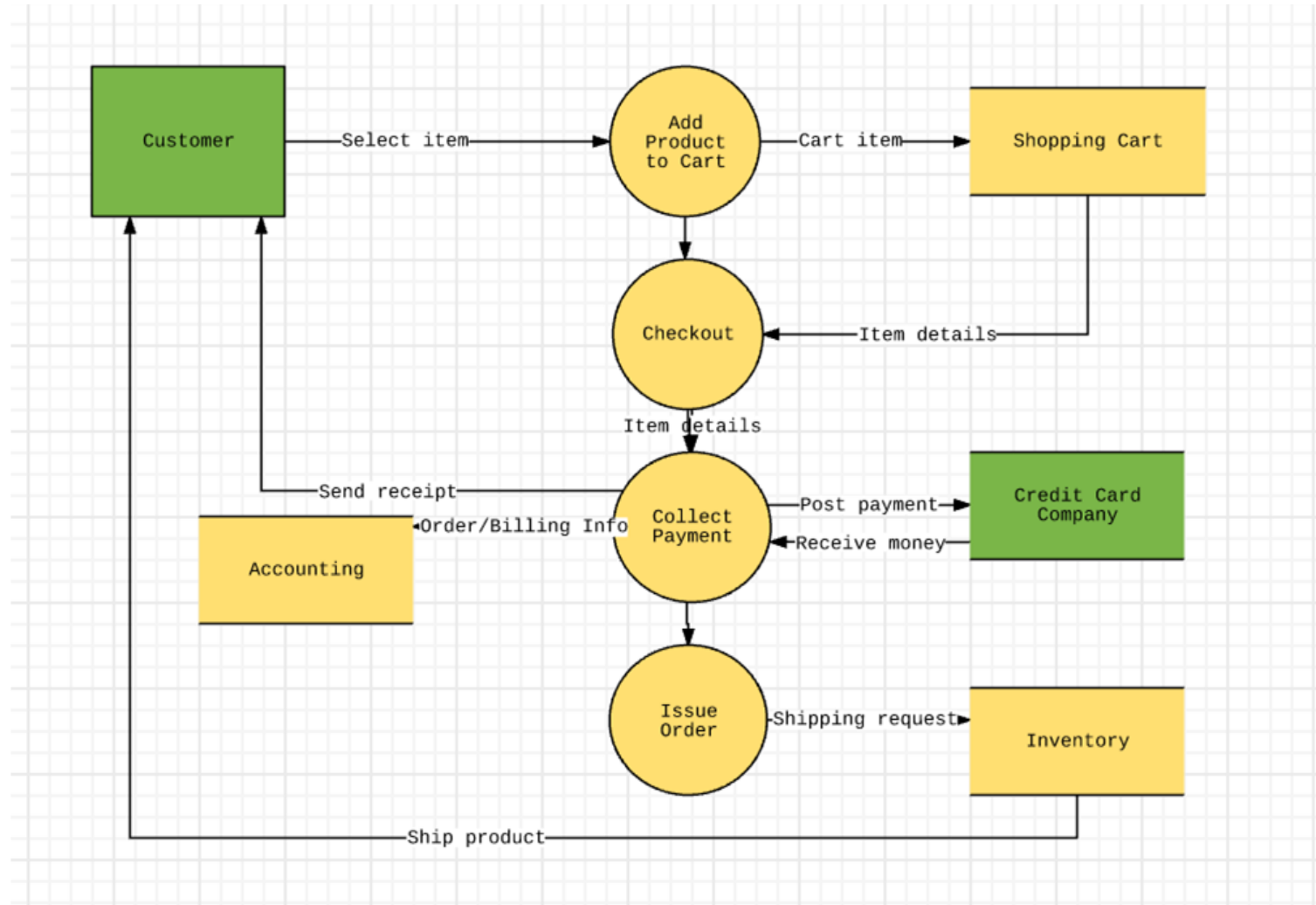
Goods sold
→

- Data flow is the path the system's information takes from external entities through processes and data stores. With arrows and succinct labels, the DFD can show you the direction of the data flow.
- A data flow represents the flow of information, with its direction represented by an arrow head that shows at the end(s) of flow connector.
- Use the type of data that is moving through the system as the name for the arrow.
- Data flows are notated as named arrows.

DFG

Remember that DFD is **not** a flow chart. Arrows in a flow chart represent the order of events; arrows in DFD represent flowing data. A DFD does not involve any order of events.

Example – Online Shopping



Rules of thumb

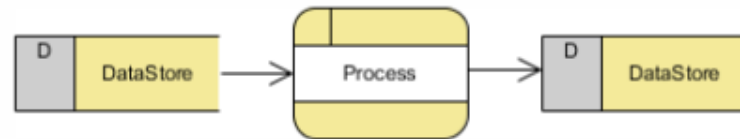
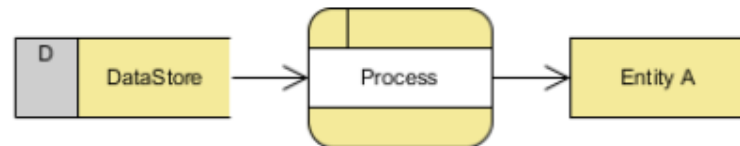
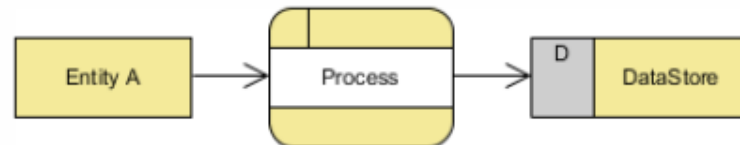
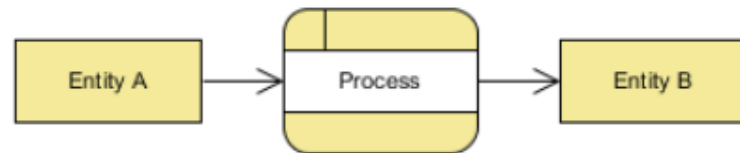
- But before you start mapping out data flow diagrams, you need to follow four rules of thumb to create a valid DFD:-
 - 1) **Each process should have at least one input and one output.**
 - 2) **Each data store should have at least one data flow in and data flow out.**
 - 3) **A system's stored data must go through a process.**
 - 4) **All processes in a DFD must link to another process or data store.**

Rules

Wrong



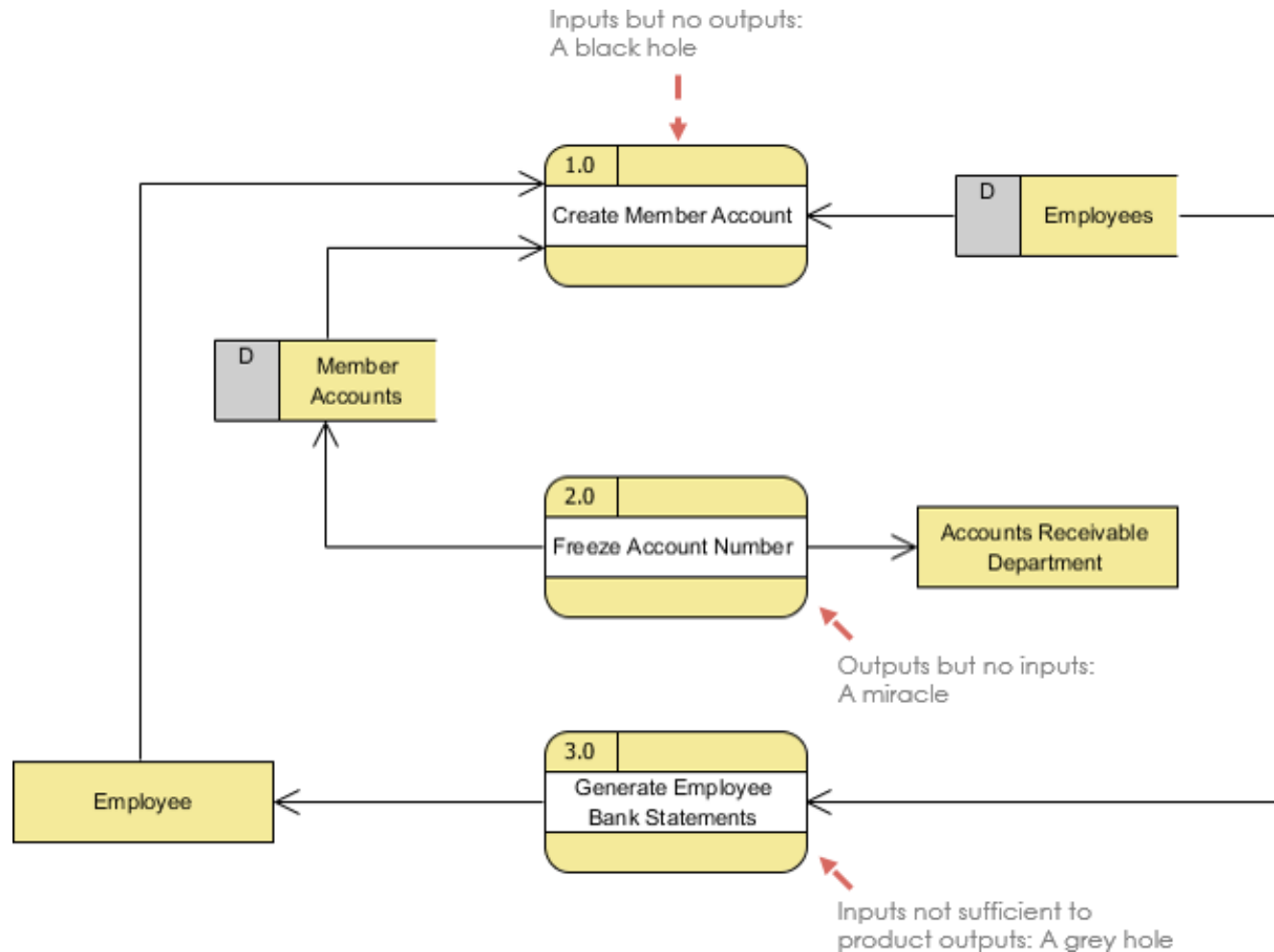
Right



Frequently Made Mistakes

- **Black holes** - A processing step may have input flows but no output flows.
- **Miracles** - A processing step may have output flows but no input flows.
- **Grey holes** - A processing step may have outputs that are greater than the sum of its inputs

Frequently Made Mistakes





UML Behavioral Model



What is UML?

- **Unified Modeling Language (UML)** is a general purpose modelling language.
- The main aim of UML is to define a standard way to **visualize** the way a system has been designed. It is quite similar to blueprints used in other fields of engineering.
- UML is **not a programming language**, it is rather a visual language.
- We use UML diagrams to portray the **behavior and structure** of a system. UML helps software engineers, businessmen and system architects with modelling, design and analysis.
- UML includes a standardized graphical notation that may be used to create an abstract model of a system: the UML model.

UML Behavioral Model

- Complex applications need collaboration and planning from multiple teams and hence require a clear and concise way to communicate amongst them.
- Businessmen do not understand code. So UML becomes essential to communicate with non programmers the essential requirements, functionalities and processes of the system.
- A lot of time is saved down the line when teams are able to visualize processes, user interactions and static structure of the system.

UML Behavioral Model

- UML is linked with **object oriented** design and analysis. UML makes the use of elements and forms associations between them to form diagrams.

Diagrams in UML can be broadly classified as:

- **Structural Diagrams** – Capture static aspects or structure of a system. Structural Diagrams include: Component Diagrams, Object Diagrams, Class Diagrams and Deployment Diagrams.
- **Behavior Diagrams** – Capture dynamic aspects or behavior of the system. Behavior diagrams include: Use Case Diagrams, State Diagrams, Activity Diagrams and Interaction Diagrams.

Object Oriented Concepts in UML

- **Class** – A class defines the blue print i.e. structure and functions of an object.
- **Objects** – Objects help us to decompose large systems and help us to modularize our system. Modularity helps to divide our system into understandable components so that we can build our system piece by piece. An object is the fundamental unit (building block) of a system which is used to depict an entity.
- **Inheritance** – Inheritance is a mechanism by which child classes inherit the properties of their parent classes.
- **Abstraction** – Mechanism by which implementation details are hidden from user.
- **Encapsulation** – Binding data together and protecting it from the outer world is referred to as encapsulation.
- **Polymorphism** – Mechanism by which functions or entities are able to exist in different forms.

Behavior Diagrams

- **Use Case Diagrams** – Use Case Diagrams are used to depict the **functionality** of a system or a part of a system. They are widely used to illustrate the functional requirements of the system and its interaction with external agents(actors). A use case is basically a diagram representing different scenarios where the system can be used. A use case diagram gives us a high level view of what the system or a part of the system does without going into implementation details.
- **Sequence Diagram** – A sequence diagram simply depicts **interaction between objects** in a **sequential order** i.e. the order in which these interactions take place. We can also use the terms event diagrams or event scenarios to refer to a sequence diagram. Sequence diagrams describe how and in what order the objects in a system function. These diagrams are widely used by businessmen and software developers to document and understand requirements for new and existing systems.

Behavior Diagrams

- **State Machine Diagrams** – A state diagram is used to represent the condition of the system or part of the system at finite instances of time. It's a behavioral diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams** . These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli.
- **Activity Diagrams** – We use Activity Diagrams to illustrate the flow of control in a system. We can also use an activity diagram to refer to the steps involved in the execution of a use case. We model sequential and concurrent activities using activity diagrams. So, we basically depict workflows visually using an activity diagram. An activity diagram focuses on condition of flow and the sequence in which it happens. We describe or depict what causes a particular event using an activity diagram.

Use Case Model

Use case

- A use case describes the sequence of actions a system performs yielding visible results. It shows the interaction of things outside the system with the system itself. Use cases may be applied to the whole system as well as a part of the system.

Actor

- An actor represents the roles that the users of the use cases play. An actor may be a person (e.g. student, customer), a device (e.g. workstation), or another system (e.g. bank, institution).

Use case diagrams

- Use case diagrams present an outside view of the manner the elements in a system behave and how they can be used in the context.

Use case diagrams comprise of –

- Use cases
- Actors
- Relationships like dependency, generalization, and association

Use case diagrams are used –

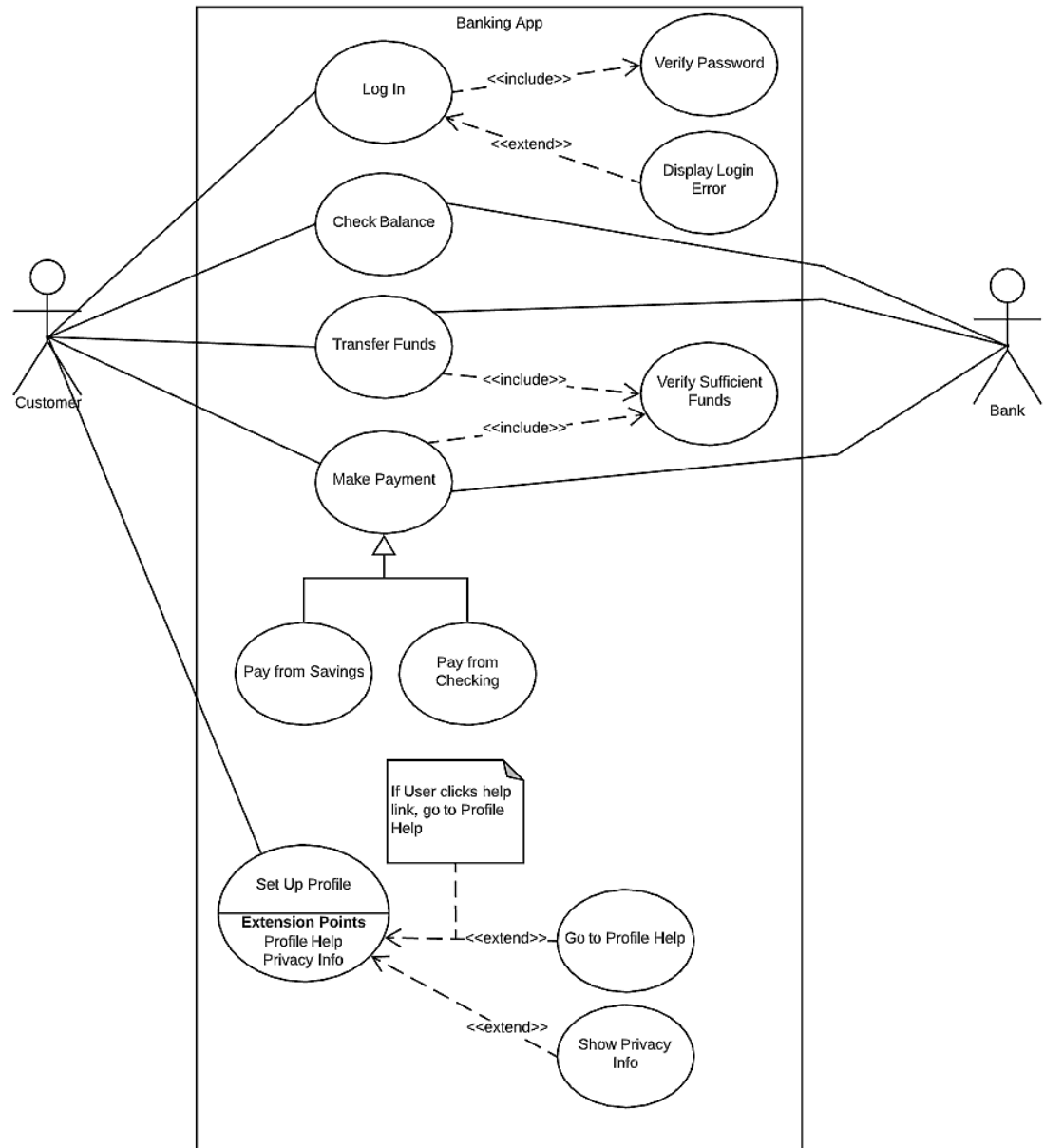
- To model the context of a system by enclosing all the activities of a system within a rectangle and focusing on the actors outside the system by interacting with it.
- To model the requirements of a system from the outside point of view.



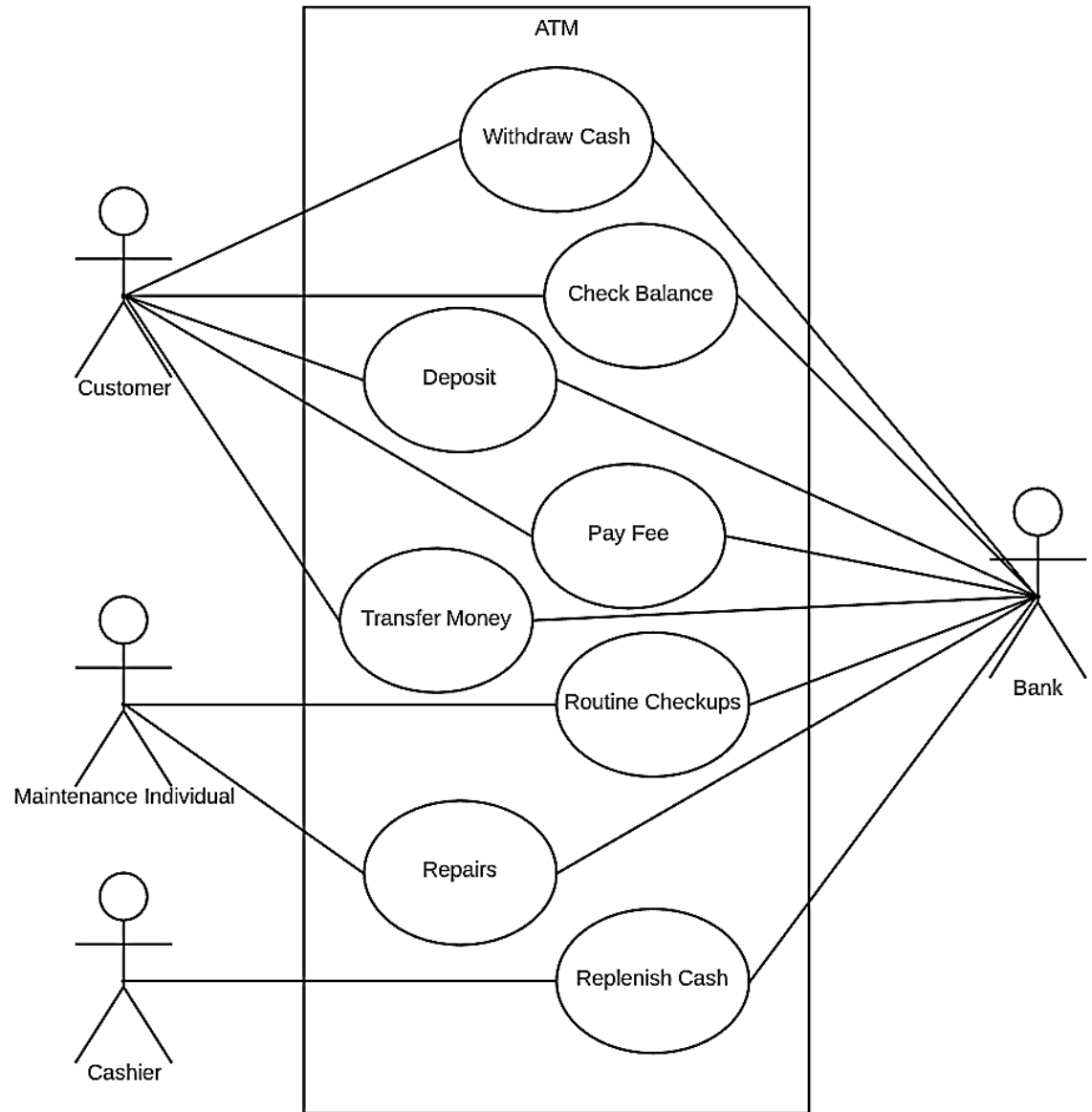
Examples



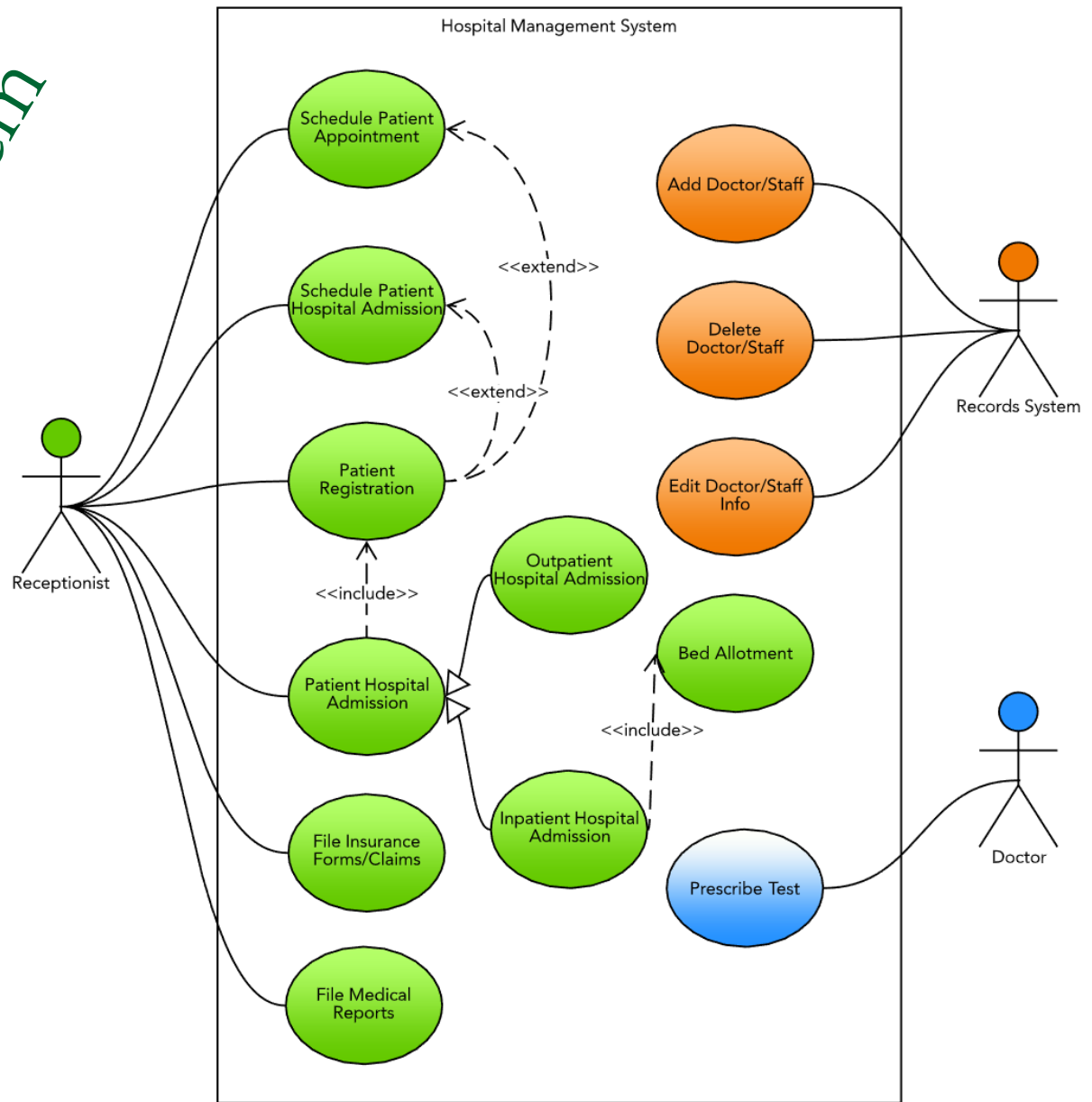
Banking App



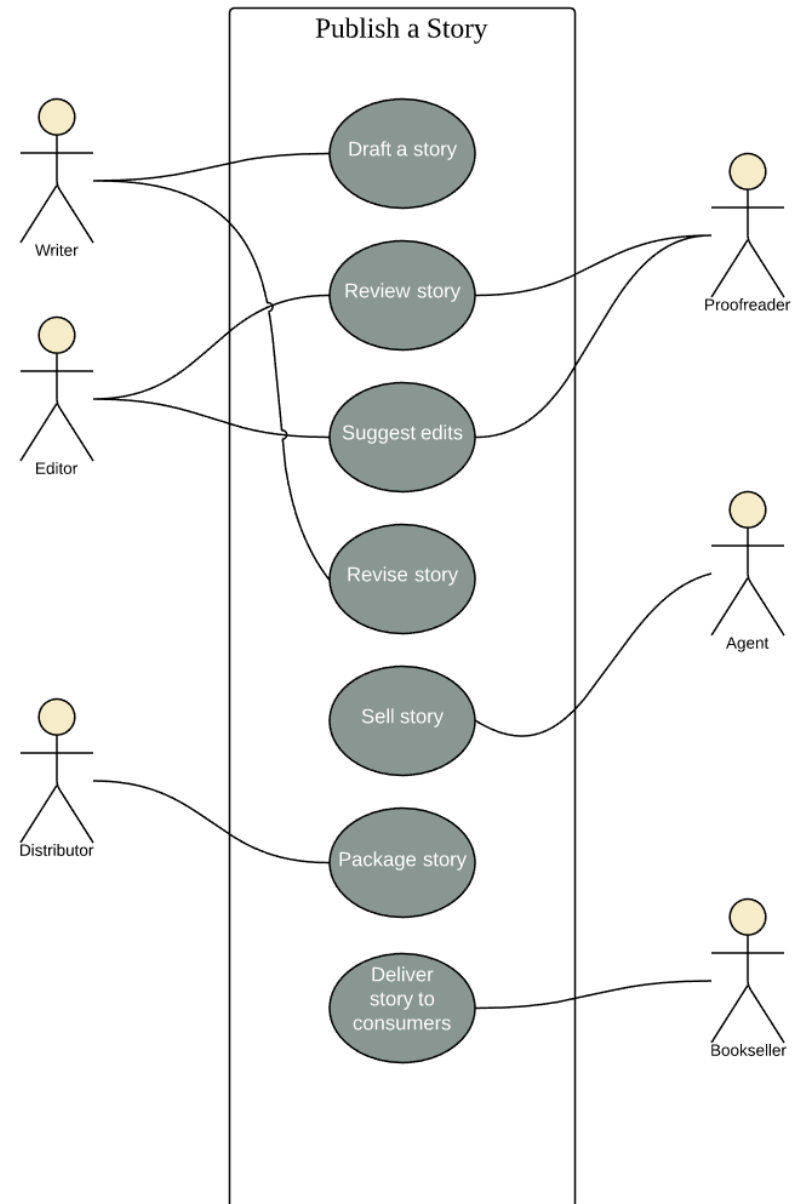
ATM



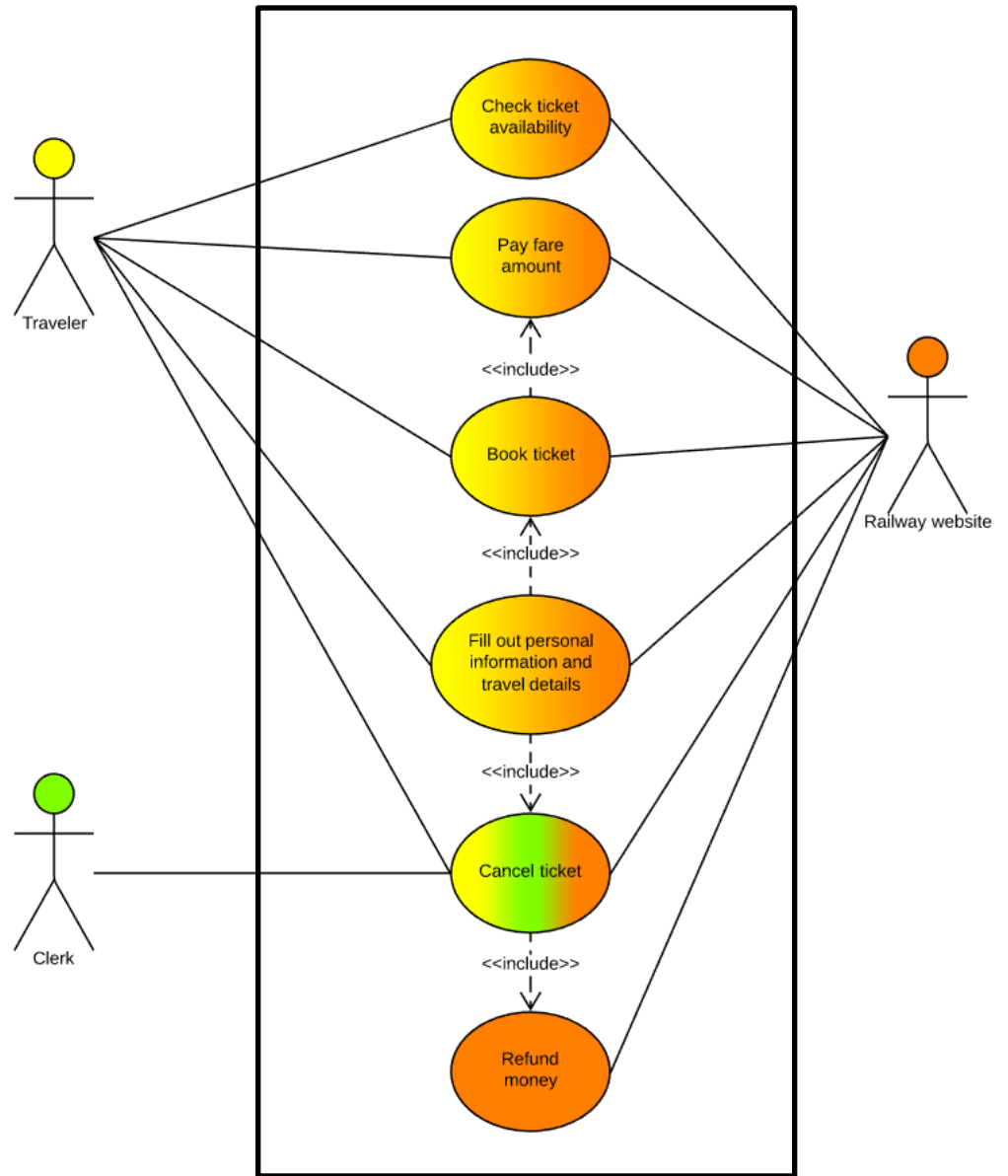
Hospital Management System



Publish a Story



Railway Reservation



Sequence Diagrams

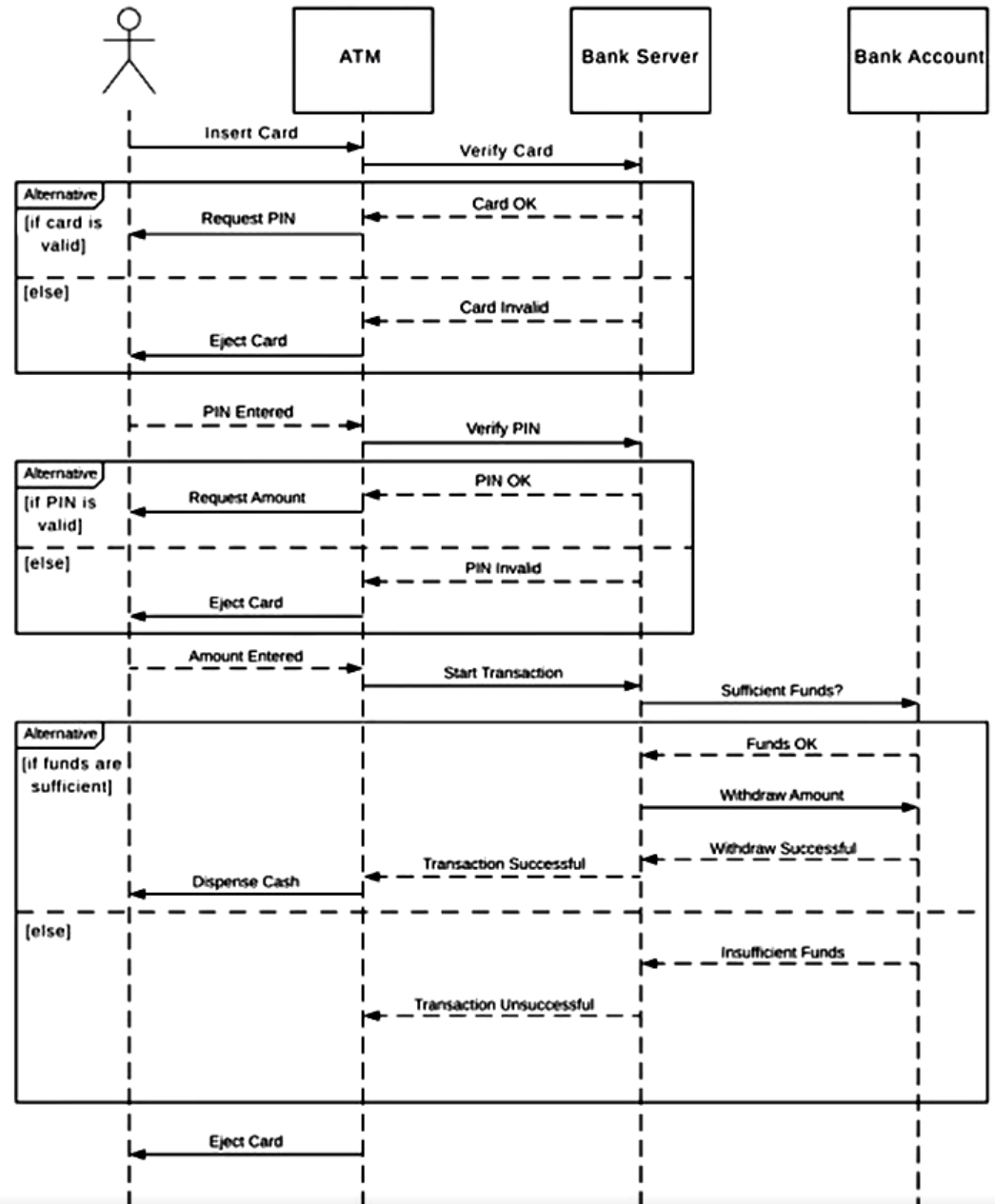
- Sequence diagrams are interaction diagrams that illustrate the ordering of messages according to time.
- **Notations** – These diagrams are in the form of two-dimensional charts. The objects that initiate the interaction are placed on the x-axis. The messages that these objects send and receive are placed along the y-axis, in the order of increasing time from top to bottom.



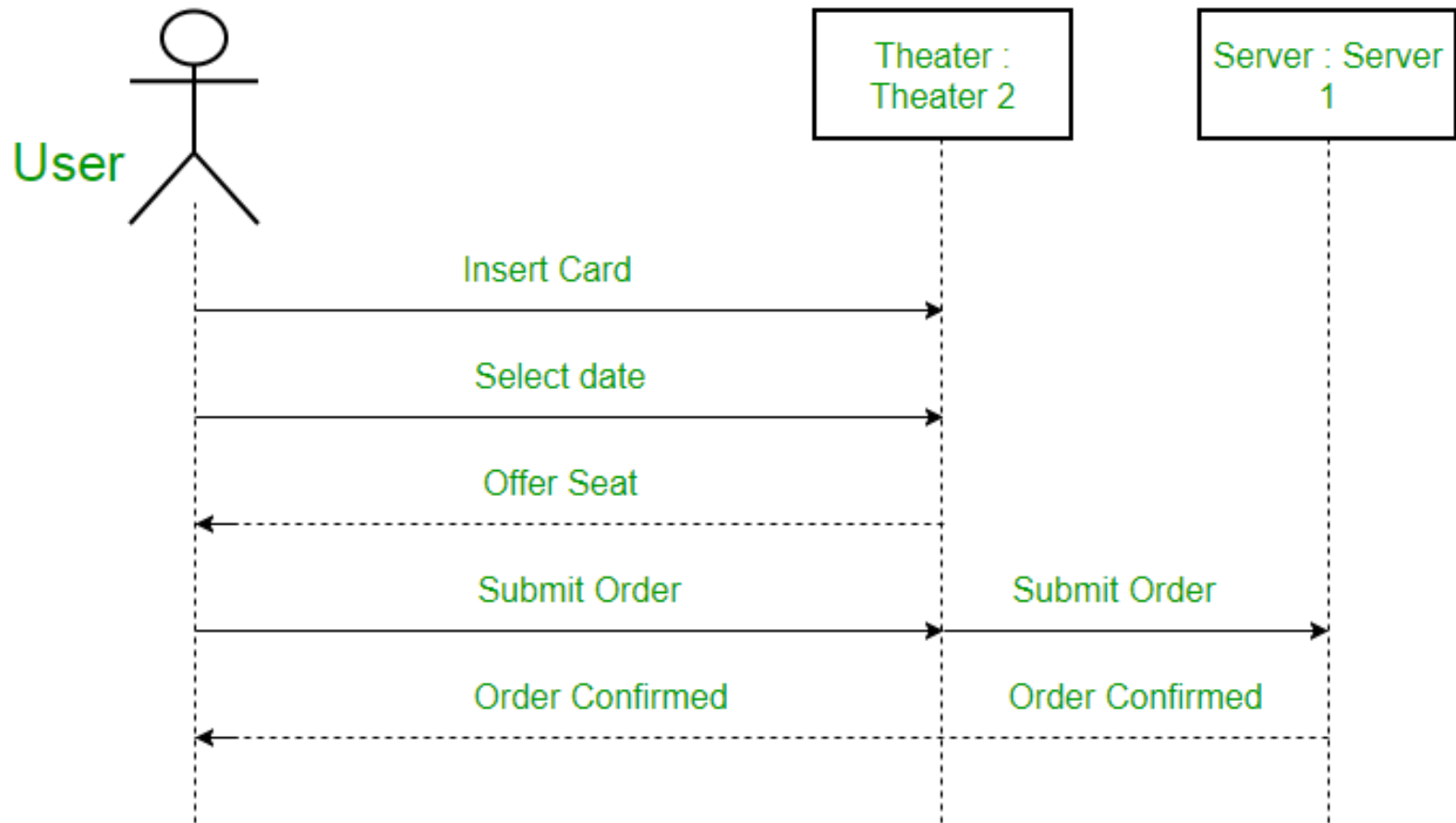
Examples



ATM Withdrawal



Seat Reservation System



Activity Diagrams

- An activity diagram depicts the flow of activities which are ongoing non-atomic operations in a state machine. Activities result in actions which are atomic operations.

Activity diagrams comprise of –

- Activity states and action states
- Transitions
- Objects

Activity diagrams are used for modeling –

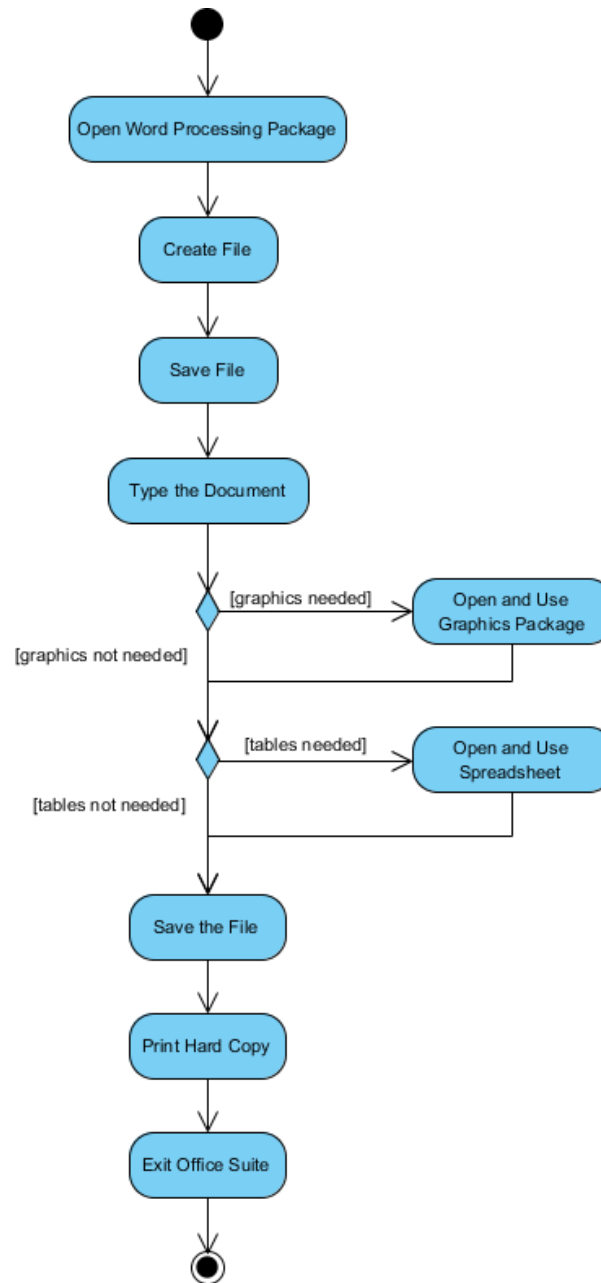
- workflows as viewed by actors, interacting with the system.
- details of operations or computations using flowcharts.



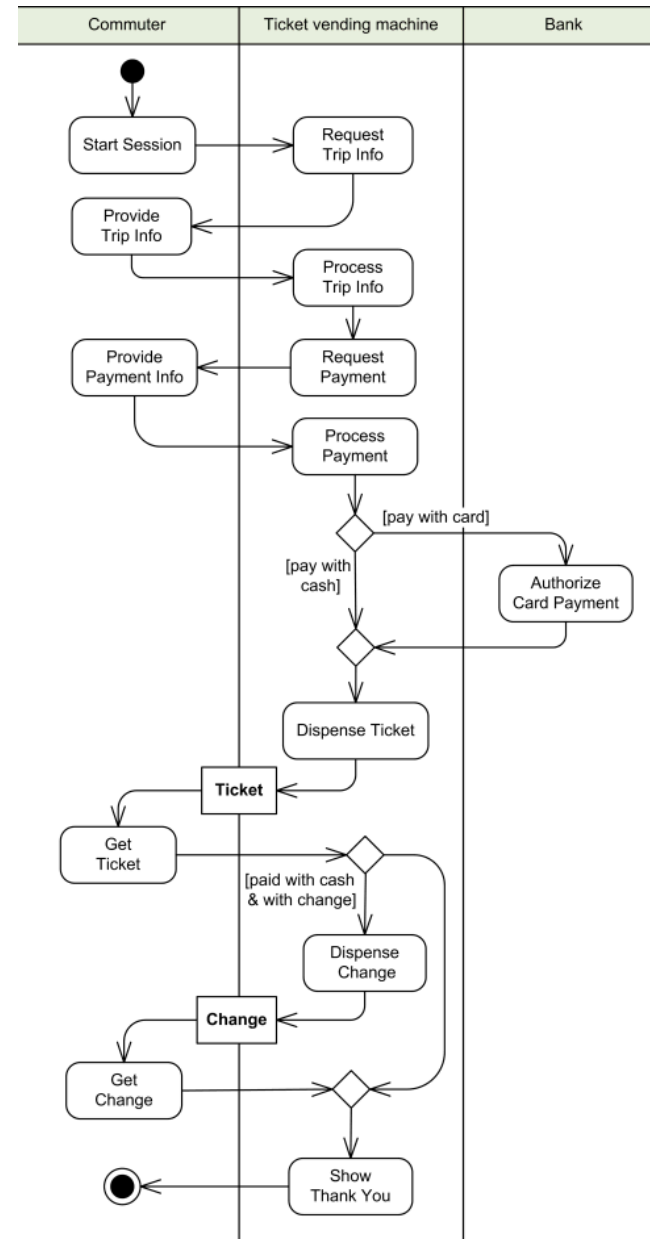
Examples



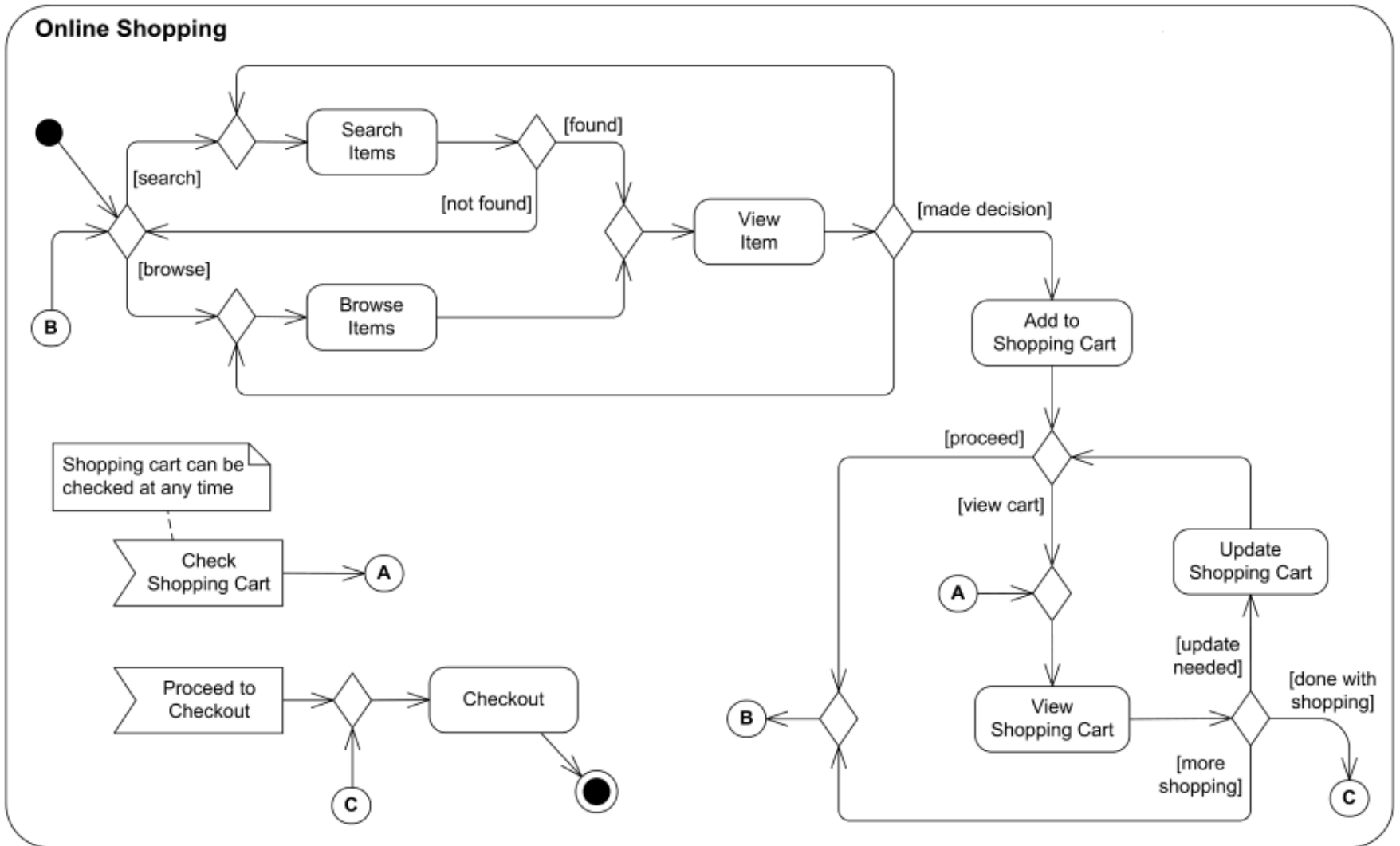
Modeling a Word Processor



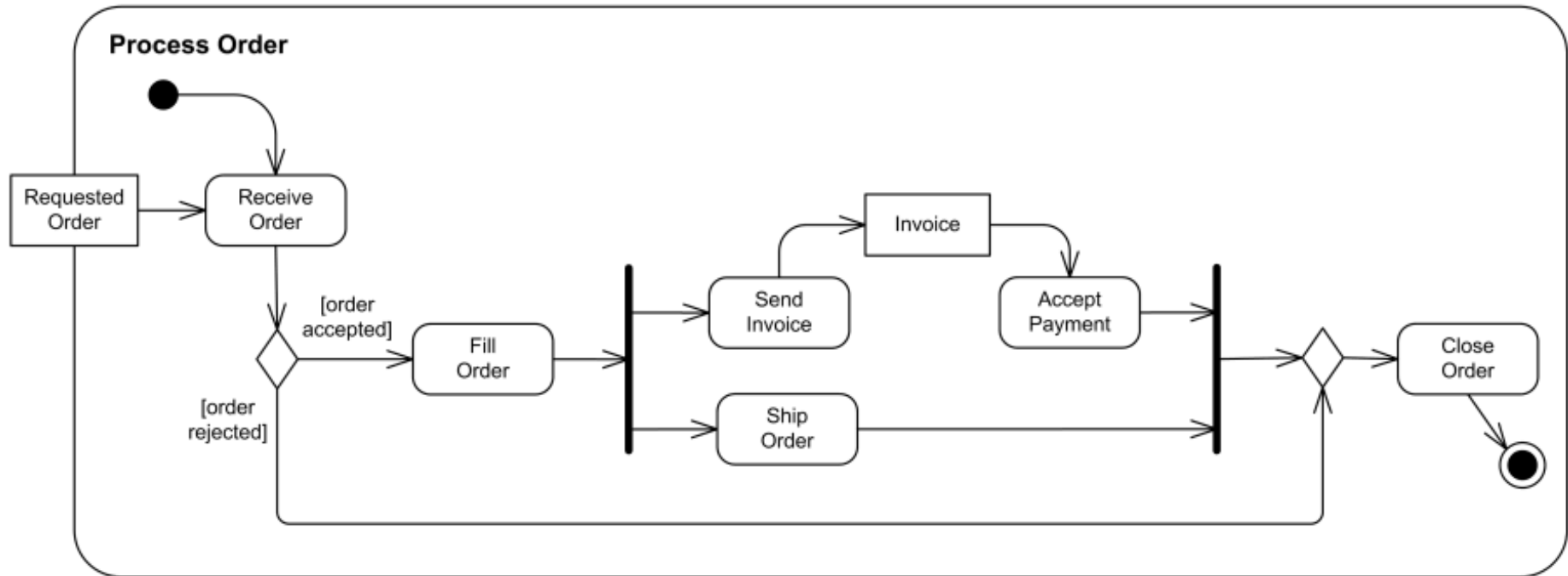
Ticket Vending Machine



Online Shopping



Process Shopping Order



State–Chart Diagrams

- A state–chart diagram shows a state machine that depicts the control flow of an object from one state to another. A state machine portrays the sequences of states which an object undergoes due to events and their responses to events.

State–Chart Diagrams comprise of –

- States: Simple or Composite
- Transitions between states
- Events causing transitions
- Actions due to the events

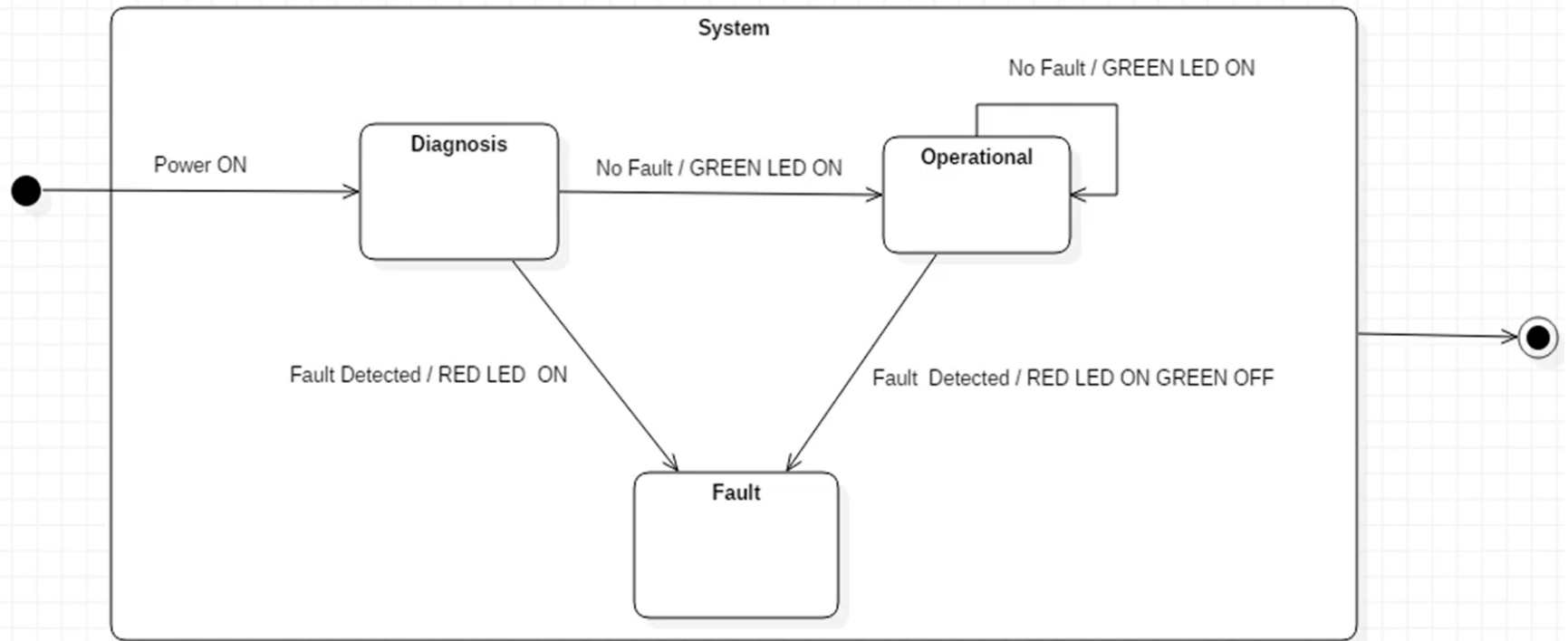
State-chart diagrams are used for modeling objects which are reactive in nature.



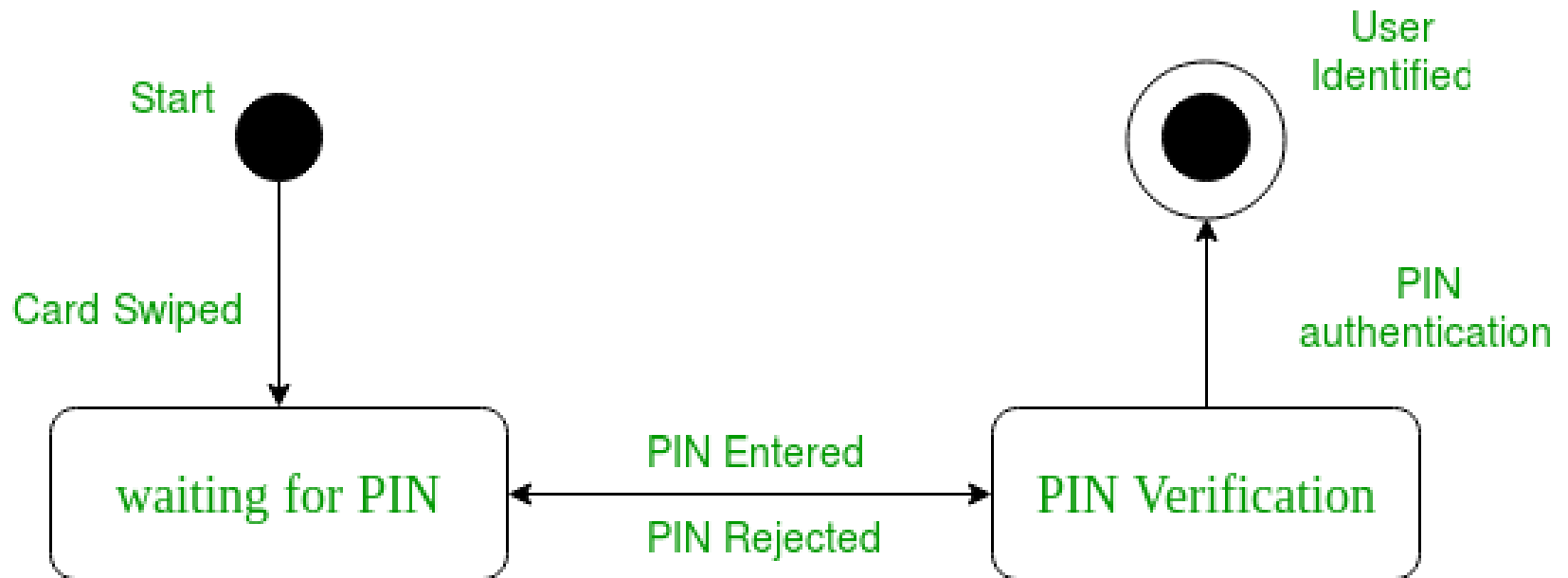
Examples



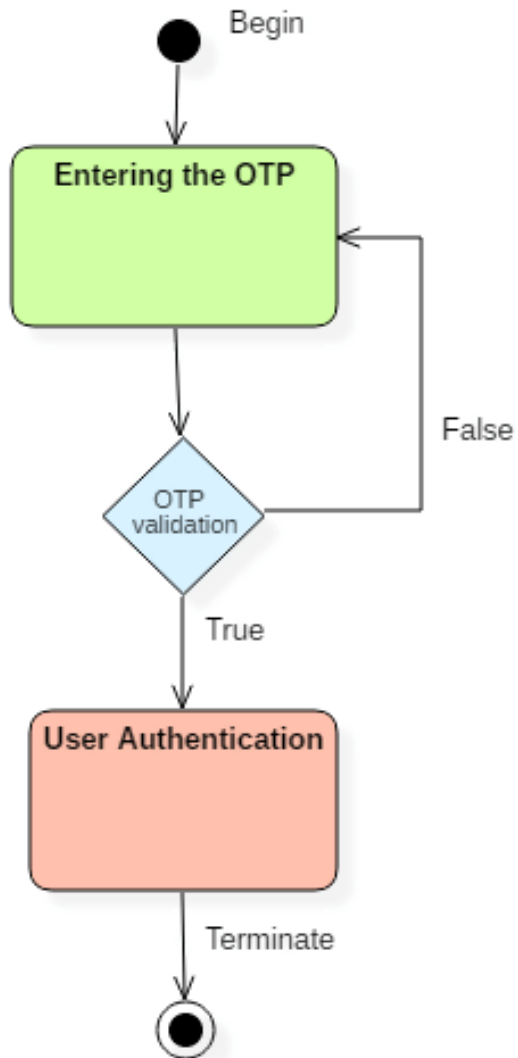
General System



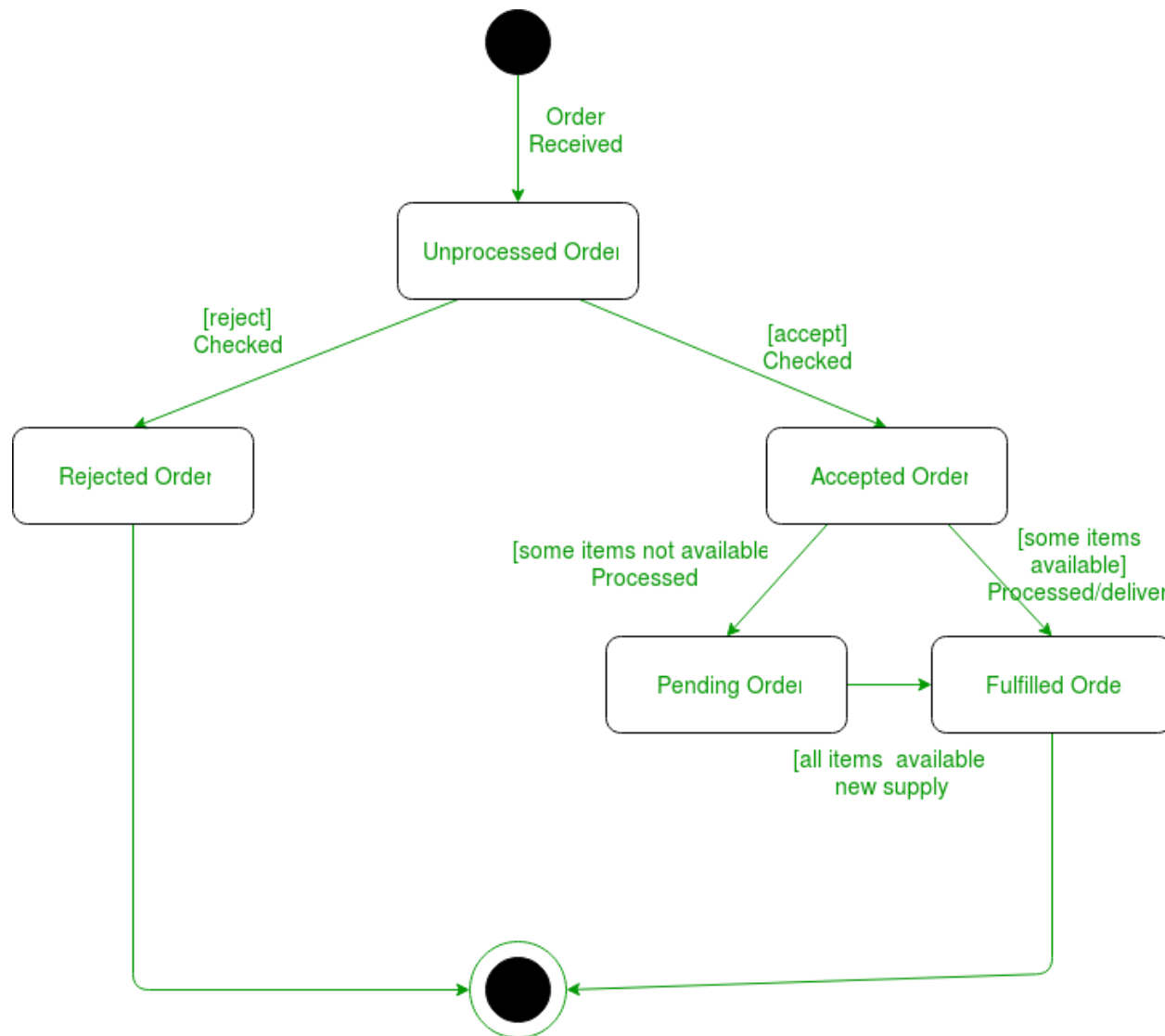
User Verification



User Verification



Online Order





Petri Net Model



Petri Nets - Definitions

- Petri Nets (PN) offer advantages because of their twofold representation: graphical and mathematical.
- It is a graphical and mathematical tool for the formal description of the logical interactions and the dynamics of complex systems. PN are particularly suited to model:
 - Concurrency
 - Conflict
 - Sequencing
 - Synchronization
 - Sharing of limited resources
 - Mutual exclusion

Petri Nets - Definitions

- A Petri net is a bipartite directed graph consisting of two kinds of nodes: places and transitions
- Places typically represent conditions within the system being modeled. They are represented by circles.
- Transitions represent events occurring in the system that may cause change in the condition of the system. They are represented by bars.
- Arcs connect places to transitions and transitions to places (never an arc from a place to a place or from a transition to a transition)

Petri Nets - Definitions

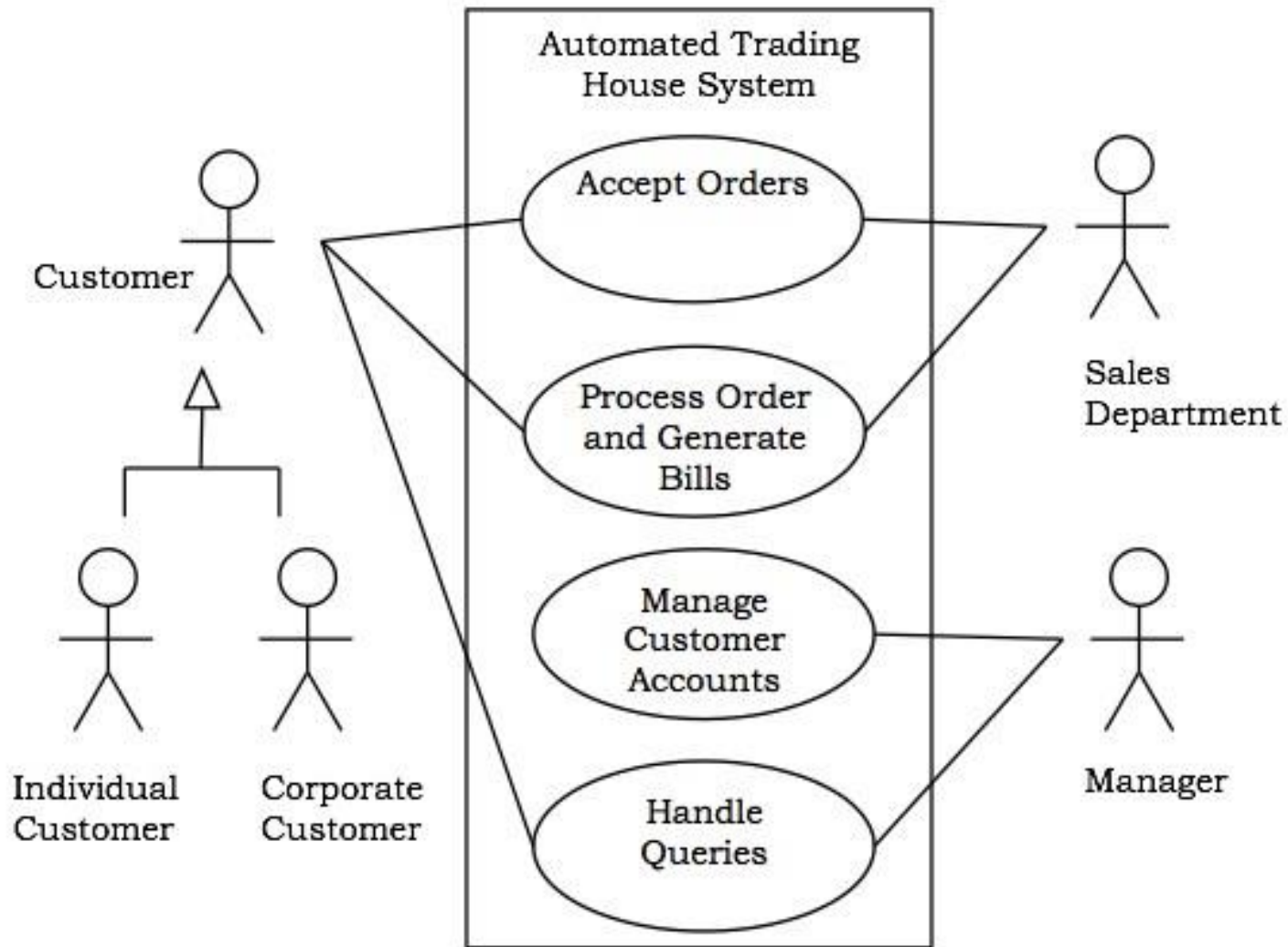
- Input arcs are directed arcs drawn from places to transitions, representing the conditions that need to be satisfied for the event to be activated
- Output arcs are directed arcs drawn from transitions to places, representing the conditions resulting from the occurrence of the event.
- Input places of a transition are the set of places that are connected to the transition through input arcs.
- Output places of a transition are the set of places to which output arcs arrive from the transition.

Use case diagrams

Example

- Let us consider an Automated Trading House System. We assume the following features of the system –
- The trading house has transactions with two types of customers, individual customers and corporate customers.
- Once the customer places an order, it is processed by the sales department and the customer is given the bill.
- The system allows the manager to manage customer accounts and answer any queries posted by the customer.

Automated Trading House System



Thank You
