Software Requirements Specification

for

Supermarket Automation System

Version 1.0

Prepared by

Ashish Bisoi

Vishal Savade

Rishab Singh

Anush Mohan

R Benjamin Franklin

List of Assignments

Sr. No.	TITLE
1	Write Problem Statement for System / Project
2	Prepare Use Case Model
3	Prepare Activity Model
4	Prepare Analysis Model-Class Model
5	Prepare a Design Model from Analysis Model
6	Prepare Sequence Model.
7	Prepare a State Model
8	Identification and Implementation of GRASP pattern
9	Identification and Implementation of GOF pattern

Assignment 1

• Introduction

· Purpose

This SRS describes the software functional and non-functional requirements for release 1.0 of the supermarket automation system(SAS). This soft:vare is designed to automate the billing and inventory system. Unless otherwise stated in all statements in a supermarket otherwise specified here are high priority and committed for release 1.0

· Document Conventions

This document for the MLA Format Guideline's. Bold-faced text has been used to emphasize section and sub-section headings. Highlighting is to point out words in the glossary and italicized text is used to label and recognize diagrams.

· Intended Audience and Reading Suggestions

This document is to be read by the development team, the project reviewers, testers and documentation writers. Our stakeholders, company manufacturing associated haidware, company providing embedded operating system and distributors who markets the finished product, may review the document to learn about the project and to understand the requirements. The SRS has been organized in order of increasinig specificity. The developers and project managers need to become intimately familiar with the SRS.

- **Product Scope •**

The-Supermarket automation software consists of the following major functions:

- Maintaining and updating the inventory of the various commodities of the supermarket.

- Creating and printing sales transaction bills. - Displaying and printing the sales statistics of various commodities for any particular period.

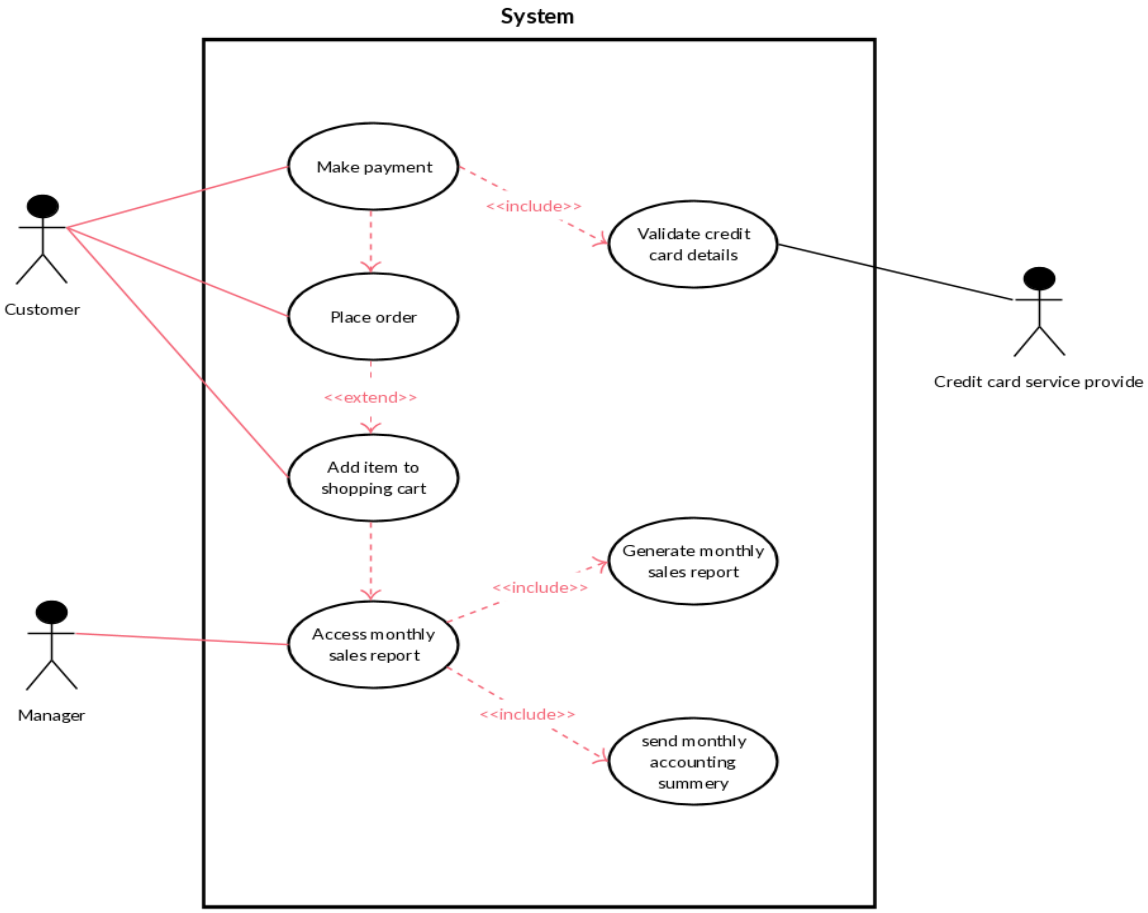
References

- SRS template by JacksonVille State University.

- en.wilcipedia.org for relevant definition

Overall Description

- **Product Perspective**



FIGUREI

The supermarket automation system is a new system that replaces the current manual processes of billing and inventory management in a supermarket. The context diagram in figure 1 illustrates the external entities and system interfaces for release 1.0. The system is expected to evolve over several releases.

- **Product Functions**

The set of functionalities that are supported by the system are documented below -

Perform Sales Transaction

Whenever any item is sold from the stock of the supermarket, this function prompts the clerk to pass the item over a bar code reader and an automatic weighing scale, the data regarding the item type and the quantity get automatically registered then. During the sales transaction, the name of the item, code number, quantity, unit price, and item price are entered into the bill. The bill indicates the total.amount payable. The inventory is then suitably updated.

Read Bar code

Input: Sold items are. passed over the reader.

Processing: Bar code of the item is read and the sold item is registered automatically.

Weigh

Input: Sold items are weighed over the automatic weighing scale.

Processing: Weight of the sold item is automatically registered.

Generation of the bill

A transaction bill containing the serial number of the sales transaction, the name of the items, quantity, unit price, item price and the total amount payable after adding the taxes is printed.

Update inventory

In order to support inventory management, this function updates the inventor Whenever an item is sold. Again, when there is a new supply arrival, an employee updates the inventory level by this function.

Cheek invertorN

The manager upon invokine this function issues query to view the inventory level by this function.

Update prices

The manager changes the price of an item by exercising this option.

Print sales statistics

This option generates a printed out sales statistics for every item the supermarket deals with.

- **User Classes and Characteristics**

2.3.1 Sales clerk :

The supermarket employs many sales clerks who are responsible for carrying out the transaction with the customers and creating and printing bills for the transactions.

2.3.2 Supermarket staff :

They are responsible for maintenance of the products in the supermarket and addition of newly arrived products to the inventory.

2.3.3 Manager :

A manager oversees the supermarket's revenue and sales functions. He views the inventory, and review and print the sales statistics.

- **Operating Environment**

The software will operate with the following software components and applications: The software product being developed will be running on Ubuntu 16.04 LTS or higher linux operating system. The minimum hardware requirement is:

- Intel Core 2 Duo/Quad/hex/Octa or higher end 64 bit processor PC or Laptop (Minimum operating frequency of 2.5GHz).
- Hard disk capacity of 1TB — 4TB.
- 64-512 MB RAM.

- **Design and Implementation Constraints**

The weighing machine in use has a certain limitation for the maximum level of weight which can be measured by it. This may constrain the accuracy of the weight involved.

It is assumed that a standard bar code reader and an automatic weighing scale is provided to the salmi clerk without which completing a sales transaction would be very difficult, The software requires a printer.

· · External Interface Requirements

· User Interfaces

Manager interface

The SAS screen displays interfaces to view the inventory, change the prices of the products, view and print sales statistics.

Sales Clerk interface

The SAS screen displays an interface to commute a transaction with a Customer. and produces and prints a bill for the transaction.

Supermarket staff interface

The SAS screen displays an interface to update the inventory for the supermarket with each arrival of new, supplies

Hardware Interfaces

For the software to function properly, the bar code reader scans the bar code from a product and sends the product ID to the software and the weighing machine sends the weight of the product.

Software Interfaces

Inventory query

· The manager queries the product whose details he/she wishes to view.

· The SAS programmatically determines the details of the product.

· The SAS displays information about the product.

· The manager selects the option to change the price of the product which updates the

corresponding price in the database.

Error handling

- The SAS may not be able to connect to the server due to error in network connection, in the case of which transaction is not possible.

4.2 Viewing sales statistics

Introduction

- The manager views the sales statistics and prints them in various formats such as pie charts, bar graphs, tabular format, etc.

Input

- Item identification parameter (such as product ID or name).
- Time period or duration.

Processing

- The SAS 16oks into the database, the cost and selling price of the particular product for every transaction in that period and generates the profit statistics in the requested format.

Outputs

- The profit statistics are displayed in the requested format for the manager, which he prints for his convenience.

4.3 Updating the prices for different commodities

Introduction

- The manager easily updates the prices for all the items available in the supermarket according to the :,hanging prices in the market.

Inputs

- The product identification parameter (such as product ID or name).
- New Price for the product.

Processing

- The SAS looks into the database and shows the product information.
- It updates the database with the new price.

Outputs

- The product information with updated price is shown.

4.4 Updating the inventory

Add to inventory

- The supermarket staff requests for the addition of the product and subsequently enters the details of the product.
- The SAS updates the product in its database and gives a confirmation message.

New transaction

- The sales clerk provides the codes of the productivity to be purchased
- On pressing the print button.. the details of the inventory are updated and a bill is produced and printed along with a confirmation message.

- **Communications Interfaces**

Any changes made to the inventory of the supermarket is automatically updated in the database which has been set up in a separate server in the supermarket itself.

4. Functional Requirements

4.1 Sales transactions

Introduction

A sale transaction both authorizes and settles the requested amount against the payment method indicated. Through authorizing, the Transaction request confirms that the payment method exists and that funds are available at the time of Authorization to cover the transaction amount.

Inputs

- Products' IDs from the bar code reader.
- Weight. reading from the automatic weighing scale.

Processing

- The SAS queries the database for the product informatiOn and calculates the total amount payable after inclusion of taxes.
- A bill is created in a printable format.

Outputs

- A formatted bill is printed for the customer.

• The supermarket staff adds new items to the inventory which have newly arrived.

Inputs

• The product ID and quantity of the product arrived,

Processing • The SAS looks into the database , if the product ID already exists in the inventory database , the quantity is updated otherwise new product information has to be added to the database.

Output

• A message id displayed confirming the update regarding the product ID and amount.

5. Non Functional Requirements

Performance

High level of performance requires high speed network and high level of connectivity Reliability

The available server must be reliable and the network connectivity in the supermarket should be proper for smooth flow of all operations and data.

Security

Every user of the software is provided a unique login ID and a password which is stored in the database hashed by SHA2 algorithm

Availability

The software is available for use from the supermarket opening time to the closing time.

5.1 inverse Requirements

• The software does not allow the inventory to be reduced form the database without the concerned item being purchased. • The software des not allow any other person except the managers to change the price of the products.

5.2 Design Constraints

No design constraints are observed in this software. 5.3Logical Database Requirements All the data is saved in the database.

5.3 Logical Database Requirements

All the data is stored in the database

• Employee information

• Product information (inventory)

• Sales information

The database allows concurrent access by various employees and is kept consistent at all the times requiring a good database desiutt.

6. Other Requirements

Each user of the SAS is required to log in his/her recount to perform differcro transactions, update inventory, view sales statistics and update processes etc. MySQL is required for maintaining the databases of inventory, sales, and employees.

Assignment 2

Aim:

Prepare Use Case Model

Theory:

UML stand for Unified Modelling Language UML is standard language for specifyin, visualising, containing and documenting the anti factor of software systems.

Goals of UML:

i) UML is a pictorial language used to make software blueprints.

ii) It can also be used to model non software systems as well as process flows in a manufacturing unit.

iii) UML has a direct relation with object oriented analysis 2 design

Object Oriented Concepts:

UML can be described as the successor of object oriented analysis and design

Following are the important concepts of object oriented world:

Objects: Objects represent an entity and the basic building block.

Class: Class is the blueprint of an object.

Abstraction: Abstraction represents the behavior of a real world entity.

Encapsulation: Encapsulation is the mechanism of binding the data together and hiding

them from the outside world.

Inheritance: Inheritance is the mechanism of making new classes from existing one.

Polymorphism: It defines the mechanism to exist in different forms.

OO Analysis and Design

Object Oriented analysis can be defined as investigation and to be more specific it is the

investigation of objects. Design means collaboration of identified objects.

So it is important to understand the OO analysis and design concepts. Now the most important purpose of OO analysis is to identify objects of a system to be designed. This analysis is also done for an existing system. Now an efficient analysis is only possible when we are able to start thinking in a way where objects can be identified. After identifying the objects their relationships are identified and finally the design is produced.

So the purpose of OO analysis and design can described as:

a. Identifying the objects of a system.

b. Identify their relationships.

c. Make a design which can be converted to executables using OO languages.

Conceptual model of UML

To understand conceptual model of UML first we need to clarify What is a conceptual model? and Why a conceptual model is at all required? A conceptual model can be defined as a model which is made of concepts and their relationships. As UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

A. UML building blocks

B. Rules to connect the building blocks

C. Common mechanisms of UML

Relationships : Articulates the meaning of the links between things.

Dependency - a semantic relationship where a change in one thing (the independent thing) causes a change in the semantics of the other thing (the dependent thing).

Notation: ----------→

(arrow-head points to the independent thing)

Association - a structural relationship that describes the connection between two things.

Notation: _____

Inheritance:

Inheritance is a relationships between a superclass and its subclasses

There are two ways to find inheritance:

-Generalization

-Specialization

Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy

Generalisation - a relationship between a general thing (called “parent” or “superclass”) and a more specific kind of that thing (called the “child” or “subclass”), such that the latter can substitute the former.

Notation:

(arrow-head points to the superclass)

Realization - a semantic relationship between two things wherein one specifies the behaviour to be carried out, and the other carries out the behaviour.

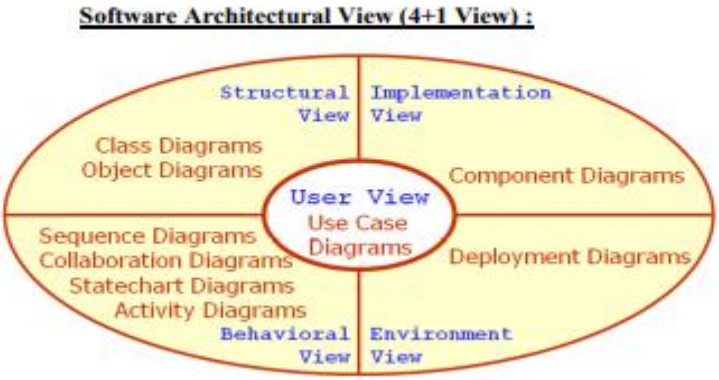
e.g. a collaboration realizes a Use Case”

the Use Case specifies the behaviour (functionality) to be carried out (provided), and the collaboration actually implements that behaviour

Notation: - - - - - - - - - -

(arrow-head points to the thing being realized)

A. Diagrams



B. Rules

Specify what a well-formed model should look like.

The UML has semantic rules for

a. Names

b. Scope

c. Visibility

d. Integrity

e. Execution

C. Common Mechanisms

a. Specifications

b. Adornments

c. Common Divisions

d. Extensibility Mechanisms

Benefits of Use Case Diagram

Use cases are the primary vehicle for requirements capture in RUP

Use cases are described using the language of the customer (language of the domain which is defined in the glossary)

Use cases provide a contractual delivery process (RUP is Use Case Driven)

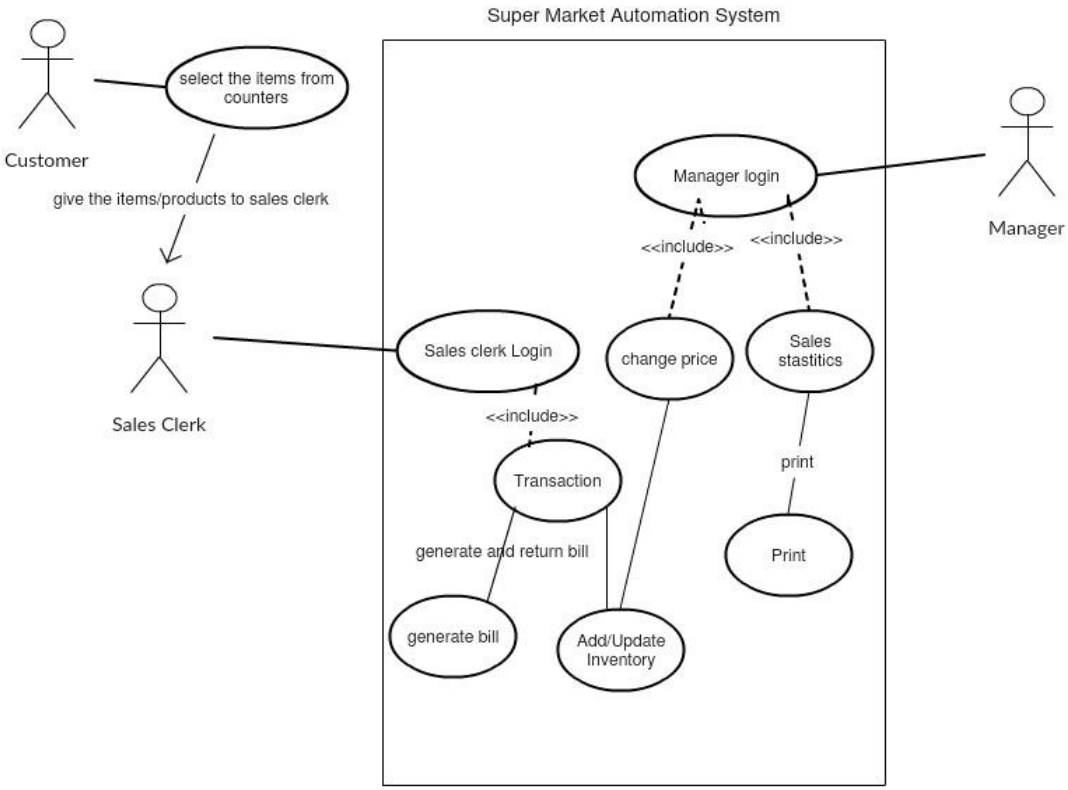
Use cases provide an easily-understood communication mechanism

When requirements are traced, they make it difficult for requirements to fall through the cracks

Use cases provide a concise summary of what the system should do at an abstract (low modification cost) level

Conclusion:

Prepared the Use case diagrams for the given system.


Super Market Automation System
Customer
select the items from counters
give the items/products to sales clerk
Sales Clerk
Manager
Manager login
<<include>>
<<include>>
Sales clerk Login
change price
Sales stastitics
<<include>>
Transaction
print
generate and return bill
Print
generate bill
Add/Update Inventory

Assignment No. 3

Aim:

To prepare the activity model

Theory:

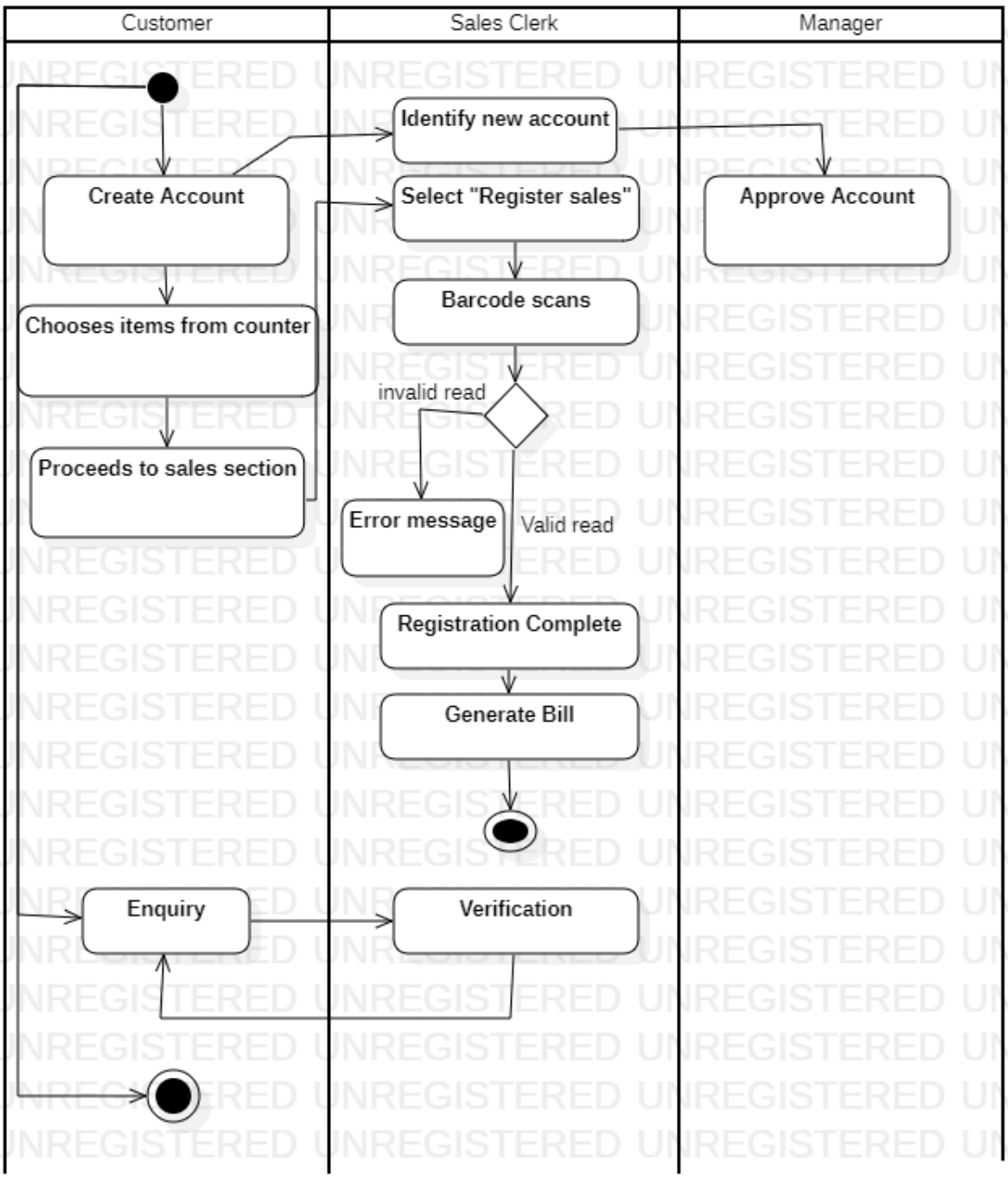
In UML, an activity diagram is used to display the sequence of activities. Activity Diagrams show the workflow from a start to finish detailing the many decision paths that exist in the progression of events contained in the activity. They may be used to detail situations where parallel processing may occur in the execution of some activities. Activity diagrams should be used in conjunction with other modeling techniques such as interaction diagrams and state diagrams. The main reason to use activity diagrams is to model the workflow behind the system being designed. Activity Diagrams are also useful for:analyzing a use case by describing what actions need to take place and when they should occur; describing a complicated sequential algorithm; and modeling applications with parallel processes. However, activity diagrams should not take the place of interaction diagrams and state diagrams. Activity diagrams do not give detail about how objects behave or how objects collaborate.

Activity diagrams can be divided into object swimlanes that determine which object is responsible for which activity. A single transition comes out of each activity, connecting it to the next activity.A transition may branch into two or more mutually exclusive

transitions. Guard expressions(inside [ ]) label the transitions coming out of a branch. A branch and its subsequent merge marking the end of the branch appear in the diagram as hollow diamonds. A transition may forks into two or more parallel activities. The fork and the subsequent join of the threads coming out of the fork appear in the diagram as solid bars.

Conclusion:

Prepared the activity diagram for the given system.


Customer
Sales Clerk
Manager
Identify new account
Create Account
Select "Register sales"
Approve Account
Barcode scans
Chooses items from counter
invalid read
Proceeds to sales section
Error message
Valid read
Registration Complete
Generate Bill
Enquiry
Verification

Assignment No. 4

Aim:

To prepare class and implement it.

Theory:

Analysis Model

1. Provide the first technical representation of a system
2. Easy to understand and maintain
3. Deals with the problem of site by partitioning the system.
4. Use graphics when possible.

Analysis Modelling include:

- Requirement analysis
- Flow oriented modelling
- Scenario based modelling
- Class based modelling
- Behavioral based modelling

Steps involved in class based modelling:

1. Identifying analysis classes
2. Defining attributes of a class
3. Defining operations of a class

These are three basic steps where Object Oriented concepts are applied and implemented. These steps are:

Object Oriented Analysis ----------> Object Oriented Design ----------> Object Oriented Implementation using Object Oriented language

Class Diagram:

The class diagram shows the building blocks of any Object Oriented design system. Class Diagrams depict the static view of the model describing what attributes and behavior it has rather than detailing the methods for achieving operations.

Class diagrams are most useful to illustrate relationships between classes and interfaces.

Class: A class is an element that defines the attributes and behaviors that an object is able to generate.

Class Notation: Classes are represented by rectangle which shows the name of the class, optionally the name of the attributes and operations.

Compartments are used to separate classnames, attributes and operations.

Generalization: Used to indicate inheritance.

Dependency: Used to model a wide range of dependent relationships between elements.

Realization: The source project implements are realises the destination realises is used to to expel the ability and compliment in a model.

Nesting: It is a connector which shows that the source element is nested within the target element.

Class details:

Rectangle

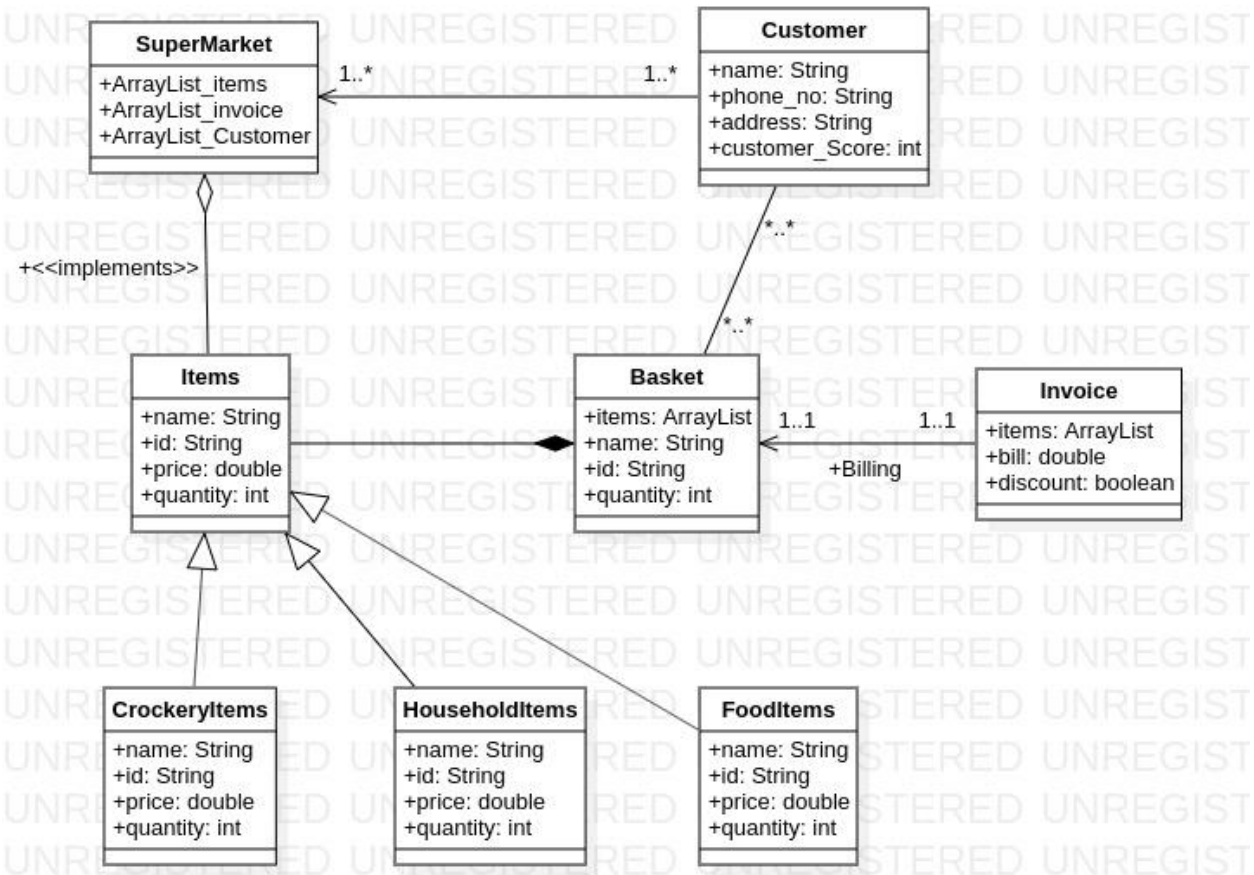
- length : double
- width : double
+ display() : void
+ remove() : void
+ setLength(new_l) : void

Interface: An interface is a specification of behaviour that implements to agree to meet its requirement.

Association: An association class is a constant construct that allows an association connection to have operation and attribute.

Conclusion:

Prepared the class diagram for the given system.


SuperMarket
+ArrayList_items
+ArrayList_invoice
+ArrayList_Customer
1..*
1..*
Customer
+name: String
+phone_no: String
+address: String
+customer_Score: int
..
..
+<<implements>>
Items
+name: String
+id: String
+price: double
+quantity: int
Basket
+items: ArrayList
+name: String
+id: String
+quantity: int
1..1
1..1
+Billing
Invoice
+items: ArrayList
+bill: double
+discount: boolean
CrockeryItems
+name: String
+id: String
+price: double
+quantity: int
HouseholdItems
+name: String
+id: String
+price: double
+quantity: int
FoodItems
+name: String
+id: String
+price: double
+quantity: int

```
package supermarketautomation;
// Super Market Interface
Interface SuperMarket{
	public void printDetails();
}
// Item dass implements SuperMarketInterface
class Items implements Supermarket{
	String id, name;
	Double price;
	int quantity,
	Items(String id, String name, Double price, int quantity){
		this.id = id;
		this.name = name;
		this.price = price;
		this.quantity = quantity;
	}
}
class CrockeryItems extends Items{
	String id, name;
	Double price;
	int quantity;
```

```
        CrockeryItems(String id, String name, Double price, int quantity){
                this.id = id;
                this.name = name;
                this.price = price;
                this.quantity = quantity;
        }
}
class FoodItems extends Items{
        String id, name;
        Double price;
        int quantity;
        FoodItems(String id, String name, Double price, int quantity){
                this.id = id;
                this.name = name;
                this.price = price;
                this.quantity = quantity;
        }
}
class Basket{
        public ArrayList<String> basket = new ArrayList<>();
        public ArrayList<Items> inventory = new ArrayList<>0;
        Basket(){}
```

```
        Basket(ArrayList<Items> e){
                basket.clear();
                inventory.addAll(e);
        }
}
class Invoice extends SuperMarketAutomation{
        public ArrayList<Items> inventory = new ArrayList<>();
        public Double bill;
        public Double discount;
        Invoid(ArrayList<Items> e, Double bill, Double discount) {
                this.inventory.addAll(e);
                this.bill= bill;
                this discount = discount;
        }
}
class Customer{
        String name, address, phone_no;
        int customer_score;
        Customer(String name, String address, String phone_no, int
customer_score){
                this.name = name;
                this.address = address;
```

```
            this.phone_no = phone_no;
            this.customer_score = customer_score;
        }
}

public class SupermarketAutomation{
        public static void main(String args[]){
            System.out.println("Inside Items Class...");
            Items I;
            System.out.println("Items Class Created");
            System.out.println("Inside CrockeryItems Class…”);
            CrockeryItems CI;
            System.out.println("CrockeryItems Class Created");
            System.out.println("Inside FoodItems Class...");
            FoodItems FI;
            System.out.println("FoodItems Class Created");
            System.out.println("Inside Basket Class... "):
            Baskets B;
            System.out.println("Basket Class Created");
            System.out.println("Inside Invoice Class...");
            Invoice Iv;
            System.out.println("Invoice Class Created");
```

```
            System.out.println("Inside Customer Class...");
            Customer C;
            System.out.println ("Customer Class Created");
        }
}
```

OUTPUT:

```
Inside Items Class...
Items Class Created
Inside CrockeryItems Class…
CrockeryItems Class Created
Inside FoodItems Class…
FoodItems Class Created
Inside Basket class...
Basket Class Created
Inside Invoice Class...
Invoice Class Created
Inside Customer Class...
Customer Class Created
```

Assignment No. 5

Aim: To prepare a design model from an analysis model.

Input: Functional resource and performance requirements.

Output: A specification providing a complete plan for implementing the system.

Techniques: Transformation, refinement, rectification, composition.

How to create a class diagram:

To create and evolve a conceptual class diagram, you need to use an iterative model.

1. **Classes:** An object is a person, place, thing, concept, event, screen or report applicable to your system classes from the main building block of an object oriented application.
2. **Responsibilities:** They are attributes and methods. Attributes are the information stored about an object whereas methods are the things objects or class do.
3. **Association:** Associations are modelled as the units combining two classes whose instances are involved in the relationship.
4. **Inheritance Relationship:** Similarity often exists among classes. You don't need to write the same code again. So inheritance is that mechanism. Inheritance model “is a” and “is like” relationship which enables you to reuse data and code easily.

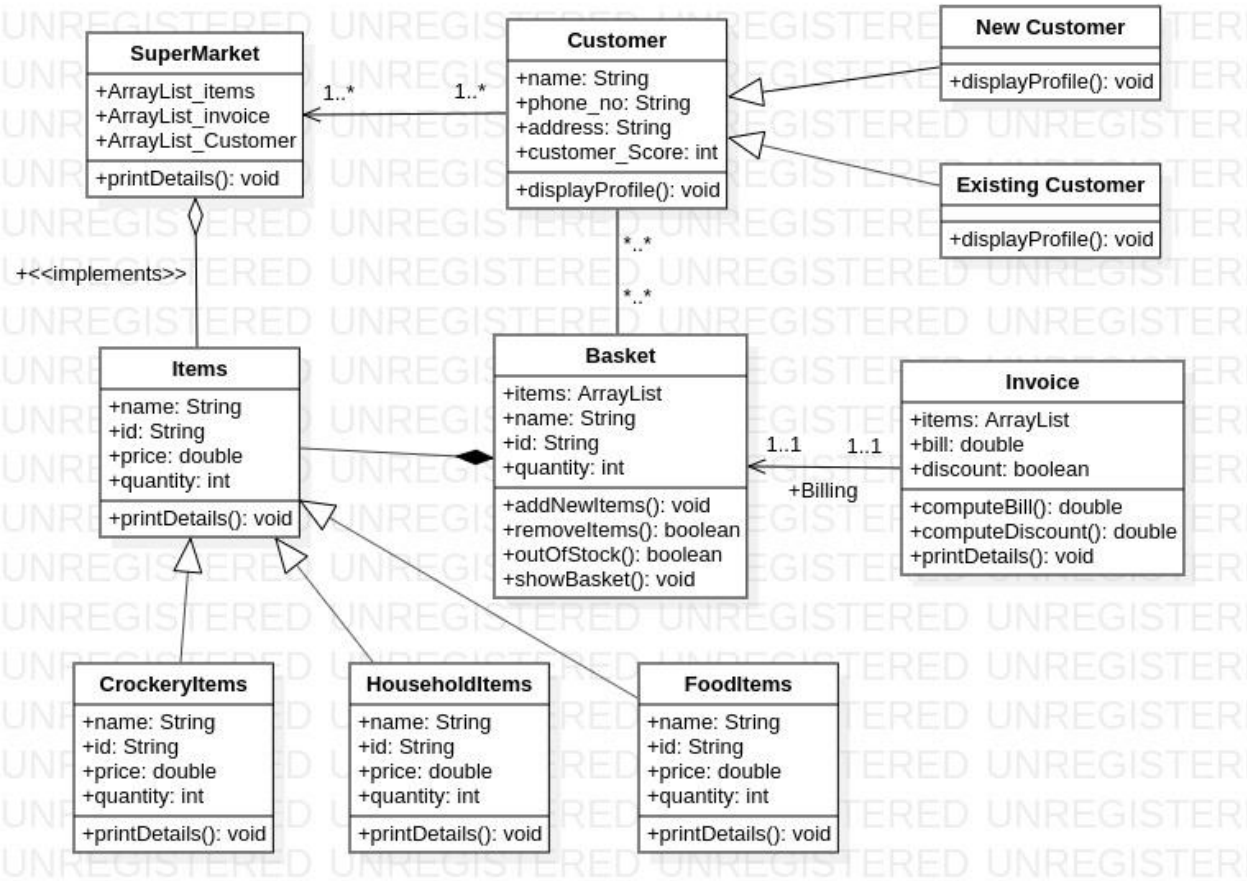
UML Class Diagrams: They show you the classes of the system, their interrelationships and operational attributes, operations of the class.

Class diagrams are typically used to though not at all need to explore domain example concepts in the form of domain models.

Diagrams are read from top to bottom and have branches and forks to describe condition and parallel activities. Forks are used to activities happening at the same time. Activity 2 and Activity 3 occurs at the same time. All the branches are followed to merge to indicate the end of conditional behaviour. After all the parallel activities must be continued by a join into the final activity state.

Conclusion:

Prepared the activity diagram for a given system.


SuperMarket
+ArrayList_items
+ArrayList_invoice
+ArrayList_Customer
+printDetails(): void
1..*
1..*
Customer
+name: String
+phone_no: String
+address: String
+customer_Score: int
+displayProfile(): void
New Customer
+displayProfile(): void
Existing Customer
+displayProfile(): void
+<<implements>>
..
..
Items
+name: String
+id: String
+price: double
+quantity: int
+printDetails(): void
Basket
+items: ArrayList
+name: String
+id: String
+quantity: int
+addNewItems(): void
+removeItems(): boolean
+outOfStock(): boolean
+showBasket(): void
1..1
1..1
+Billing
Invoice
+items: ArrayList
+bill: double
+discount: boolean
+computeBill(): double
+computeDiscount(): double
+printDetails(): void
CrockeryItems
+name: String
+id: String
+price: double
+quantity: int
+printDetails(): void
HouseholdItems
+name: String
+id: String
+price: double
+quantity: int
+printDetails(): void
FoodItems
+name: String
+id: String
+price: double
+quantity: int
+printDetails(): void

// CODE

```
package supermarketautomation;


import java.util.ArraybList;
import java.util.S8canner;


public class superMarketAutomation {
        public static Basket cart;
        public static Customer customer;
        public static Items items;
        public static Invoice invoice;
        public static ArrayList<String> cart_list;
        public static ArrayList<Items> inventory list;
        static Scanner s = new Scanner (System. in);
        public static void main (String[] args) {
                //Initialisation
                CrockeryItems crockeryItems = new CrockeryItems () ;
                crockeryItems.addCrockeryItem() ;
                HouseholdItems householdItems = new HouseHoldItems();
                householdItems.addHouseHoldItem();
                FoodItems fooditems = new FoodItems () ;
                fooditems.addFoodItem();
                items = new Items();
                items. appendItems (crockerItems.crockery) ;
                items. appendItems (householdItems.household) ;
```

```
                items .appendItems (fooditems.food) ;
                //items.printDetails();
                //Customer Interaction part
                customer = new Customer ();
                System.out.print ("WELCOME TO AIT SUPERMARKET
\n-------- \n1.New Customer\n2.Exisiting Customer\n--->") ;
                int option = s.nextInt();
                s.skip("\n");
                System.out.print ("Enter Details\nName:") ;
                customer.name = s.next();
                System.out.print("PhoneNo: ");
                customer.phone_no = s.next();
                System.out.print ("Address: ");
                customer.address = s.next();
        }
}
class Invoice extends SuperMarketAutmation{
        public void printDetails() {
                System.out.printf("\n\nINVOICE: \n----\n%-10s%-20s%-
10s%-10s", "ID", “NAME", "QUANTITY", "PRICE");
                int sum = 0;
                for (String t: Start list) {
                        for (Items e: inventory list) {
                                String a1 = t.split("\\+") (0) .toString();
                                String a2 = t.split("\\+") [0] .toString();
                                if (a1.equals(e.id)) {
```

```
                                        System.out.print ("\n%-10s%-20s%-
10s%-10.2f", a1.e.name, a2.Integer.parseInt (a2) *e.price);
                                        sum += Integer.parseInt (a2)*e.price;
                                }
                        }
                }
                System. out .print1n("\n---------------------------------\nTOTAL:Rs"+sum) ;
                }
}
class Customer {
        String name, address, phone_no;
        int customer_score;
        public void displayProfile() {
                System.out.print ('\n\nCUSTOMER DETAILS\n----
\nName : %-15s\nPhoneNo: %-15s\nCustomer Score: %-5d\nAddress: %-
29s", name, phone_no, customer score, address);')
        }
}
public ArrayList<String> basket = new ArrayList<>();
public ArrayList<Items> Inventory = new ArrayList<>();
public void addNewItem(String e, int q){
        if(outOfStock (e, q))
                basket .add(e4"4"4q) ;
        else{
        System. out.println("\nOUT OF STOCK NOT AVAILABLE\n");
        }
```

```
}
public boolean removeItems(Items e) {
        if (basked.contains (e) )
                basket. remove (e) ;
        return true;
}
public boolean outofStock (String e, int q){
        for(Items i: inventory) {
                if((i.id) .equals(e)) {
                        if(i.quantity < q)
                                return true;
                        else
                                return false;
                }
        }
        return true;
}
public ArrayList<String> fetchList() {
        return basket;
}
public ArrayList<Items>fetchInventory() {
        return Inventory;
}
public void showBasket (} {
        System.out.print("\n\nBASKET ITEMS \n%-10s%-10s", "ITEM ID") ;
}
```

```
Class Householditems extends items {
        public ArrayList<Items> household = new ArrayList<Items>();
        public void addHouseholditems () (
                household. add (new Items ("1H", "Chair", 20.00, 30));
                household. add (new Items ("2F", "Burger", 80.00, 12));
                household. add (new Items ("1H", "Peanuts", 50.00, 20));
                household. add (new Items ("1H", "Noodles", 170.00, 50));
                household. add (new Items ("1H", "Chocolate", 300.00, 13));
}
        public void PrintDetails() {
                for (items i: household) {
                        System.out.printf("%-10s%-20s%-10.2s%-10d\n", i.id, i.name, i.price,
                i.quantity);
                }
        }
}
Class FoodItems extends Items {
        public ArrayList<Items> food new ArrayList<Items> () ;
        public void addFoodItem() {
                food.add (new Items ("1F", "Chips", 20.00, 30));
                food.add (new Items ("2F", "Burger", 80.00, 12));
                food.add (new Items ("3F", "Peanuts", 50.00, 20));
                food.add (new Items ("4F", "Noodles", 170.00, 50));
                food.add (new Items ("5F", "Chocolate", 300.00, 13));
        }
        public void printDetails() {
```

```
                for (Items i: food) {
                        System.out.printf("%-10s%-20s%-10.2s%-10d\n",
i.id, i.name, i.prince, i.quantity);
                }
        }
}
Class Basket () {
        Double price;
        int quantity;
        Items () {}
        Items (String id, String name, Double price, int quantity){
                this.id = id;
                this.name = name;
                this.price = price;
                this.quantity = quantity;
        }
        public void appeandItems (ArrayList<Items> a){
                items.addAll (a);
        }
        public ArrayList<Items> returnList () {
                return items;
        }
        @Override
        public void printDetails() {
                System.out.printf ("%-10s%-20s%-10s%\n", "ID",
"NAME", "PRICE", “QUANTITY") ;
```

```
                for(items i: items) {
                        System.out.printf ("%-10s%-20s%-10.2f%-10d\n", i.id, i.name, i.price,
                i.quantity);
                }
        }
}
class Crockeryitems extends Items{
        public ArrayList<Items> crockery = new ArrayList<Items>();
        public void addCrockeyItem() {
                crockery.add (new Items("1C", "Saucers set", 120.00, 10));
                crockery.add (new Items ("2c", "Cup set", 120.00, 10));
                crockery.add (new Items ("3C", "Tea Pot", 120.00, 10));
                crockery.add (new Items ("4C", "Milk Jug", 120.00, 10));
                crockery.add (new Items ("5c", "Pepper set", 120.00, 10));
        }
        public void printDetails() {
                for (items i: crockery) {
                        System.out.printf ("%-10s%-20s%-10.2f%-10d\n", i.id, i.name, i.price
        , i.quantity);
                }
        }
        cart = new Basket (items.returnList ());
        Systems.out.print("\n\n+HELLO" + customer.name + "\nBasket
Items: 0\nBill Amount Rs.00") ;
        addToCart ();
        while (true) {
```

```
                System. out. print ("App ITEMS ? (Yes/No)");
                if(s.next ().equals ("Yes")){
                        addToCart();
                }
                else{
                        break;
                }
        }
        cart_list = cart.fetchList ();
        inventory_list = cart.fetchInventory();
        invoice = new Invoice ();
        invoice.printDetails();
}
public static void addToCart () {
        System.out.print("\n\nINVENTORY\n ----- \n");
        items.printDetails();
        System.out.print ("\n--> ");
        String id selected = s.next();
        System.out.print("Qunatity: ");
        cart.addNewItems(id _ selected, S.nextInt ());
}
//SuperMarket Interface
interface SuperMarket {
        public ArrayList<Items> items = new ArrayList<items>();
        public ArrayList<Invoice> invoices = new ArrayList<Invoice>();
        public ArrayList<Customer> customers = new ArrayList<Customer>();
```

```
        public void printDetails();
}
//item class implements SuperMarketInterface
class items implements SuperMarket {
        String id, name;
}
```

ASSIGNMENT NO. 06

Aim : To prepare diagram collaboration and sequence

Objective : To learn behavioral model and creation of sequence diagram.

Theory : 1) About sequence and collaboration.

a. Contents

b. Common Users

Input : Refer UML user guide –Rumbaugh for Object content analysis and design (OOAD).

A sequence diagram is a form of interaction diagram which shows object and lifeline running down the page and with their interactions over time represented as message drawn from source lifeline to target lifeline. Sequence diagram are not intended for showing complex procedural logic.

Lifeline : It represent the Individual participant in as sequence Diagram. It will usually have a rectangle combining its object name.

Message : They are displayed as arrows. Message can be complete lost or found , synchronous or asynchronous , call or signal.

Self-Message : It can represent a recursive call of an operation or one method calling another method belonging to the same object. It is creating a nested focus of control in the lifeline execution occurrence.

Lost and Found Messages : Lost are those that are time sent but don't arrived at the intended recipient, or which go to recipient but not shown. Found are those that arrived from an unknown sender.

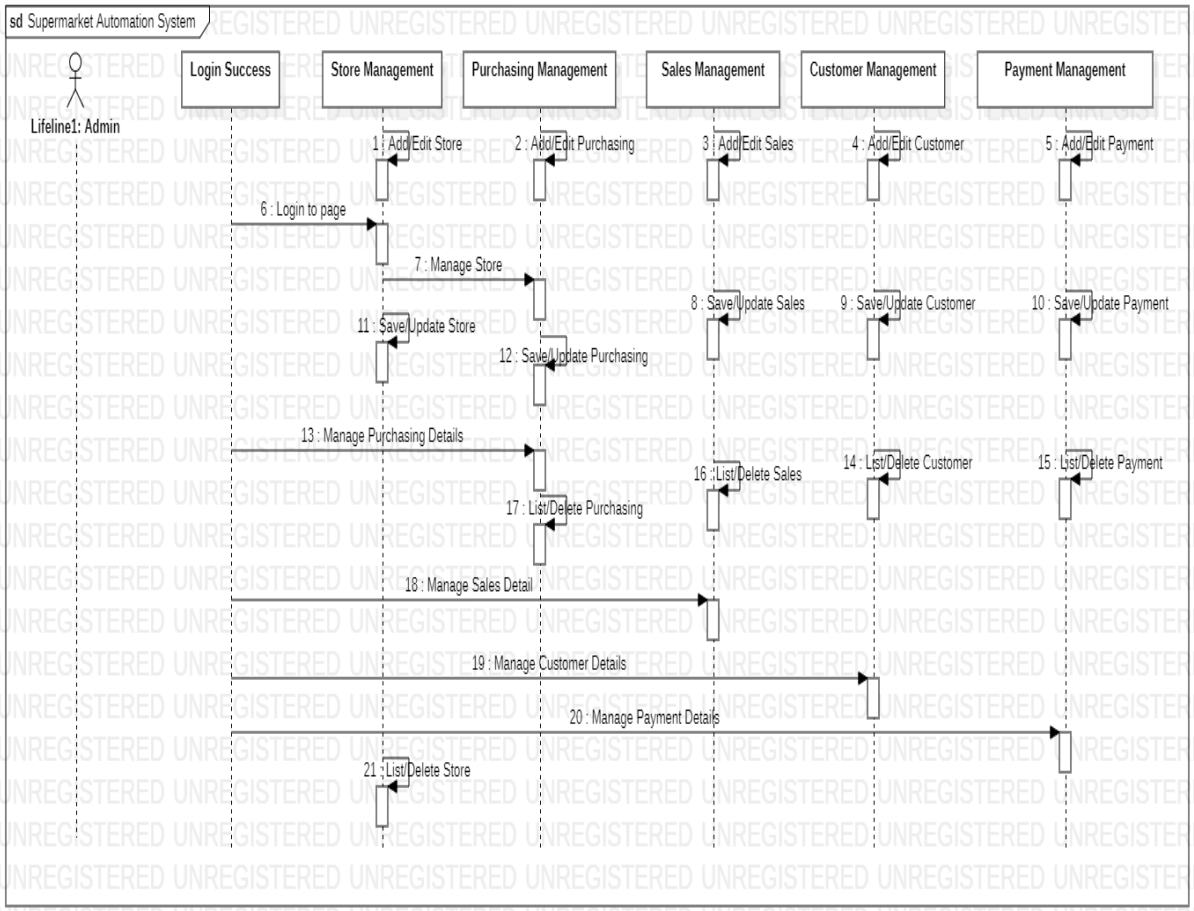
Duration and Time constraint : By default a message shown as a horizontal line, because the lifeline represents the passage of time down of a message , the message will be shown a sloping curve.

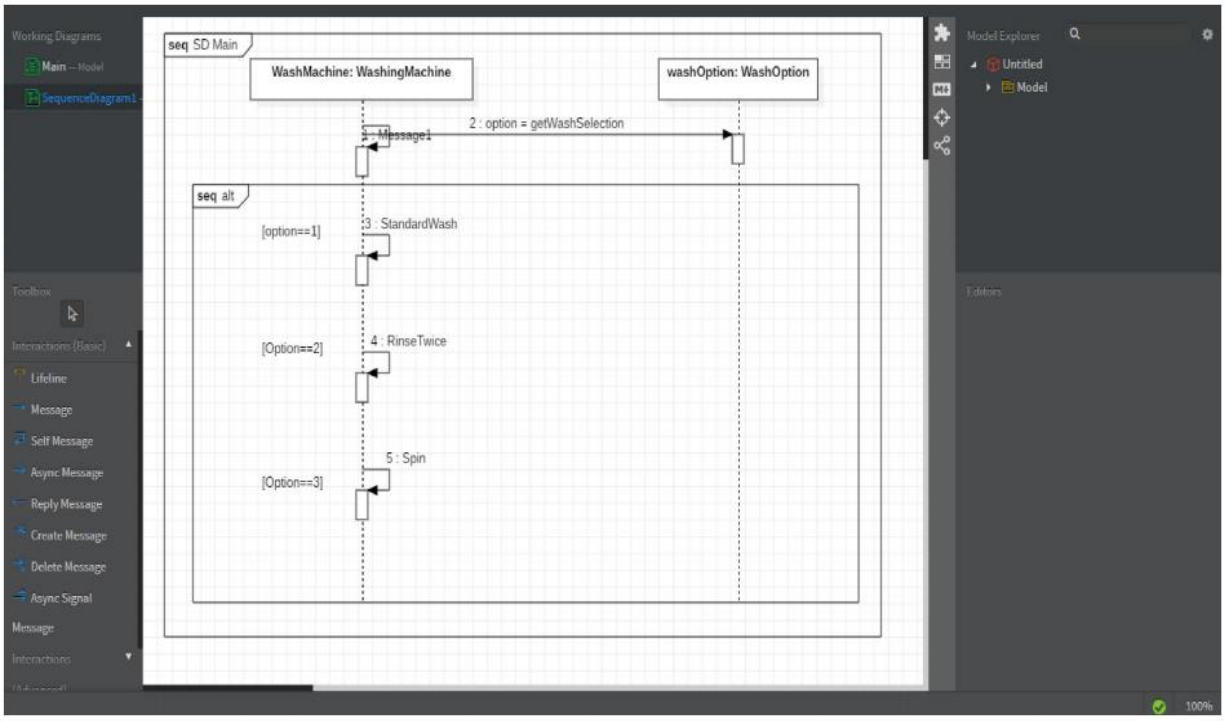
Gate : It is a connection point for connecting a message inside a fragment with message outside a fragment.

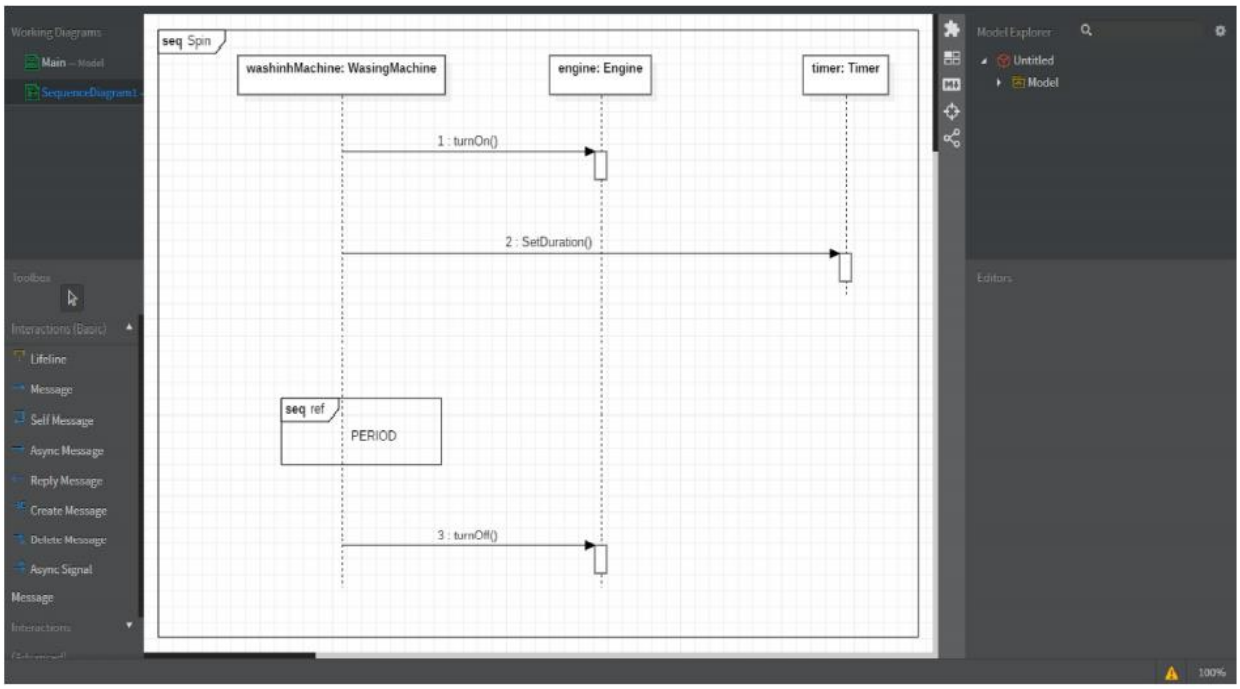
State Invariant / Continuation : It is a constraint placed on outline of lifeline that must be true at runtime. It is shown as rectangle with semi-circular ends.

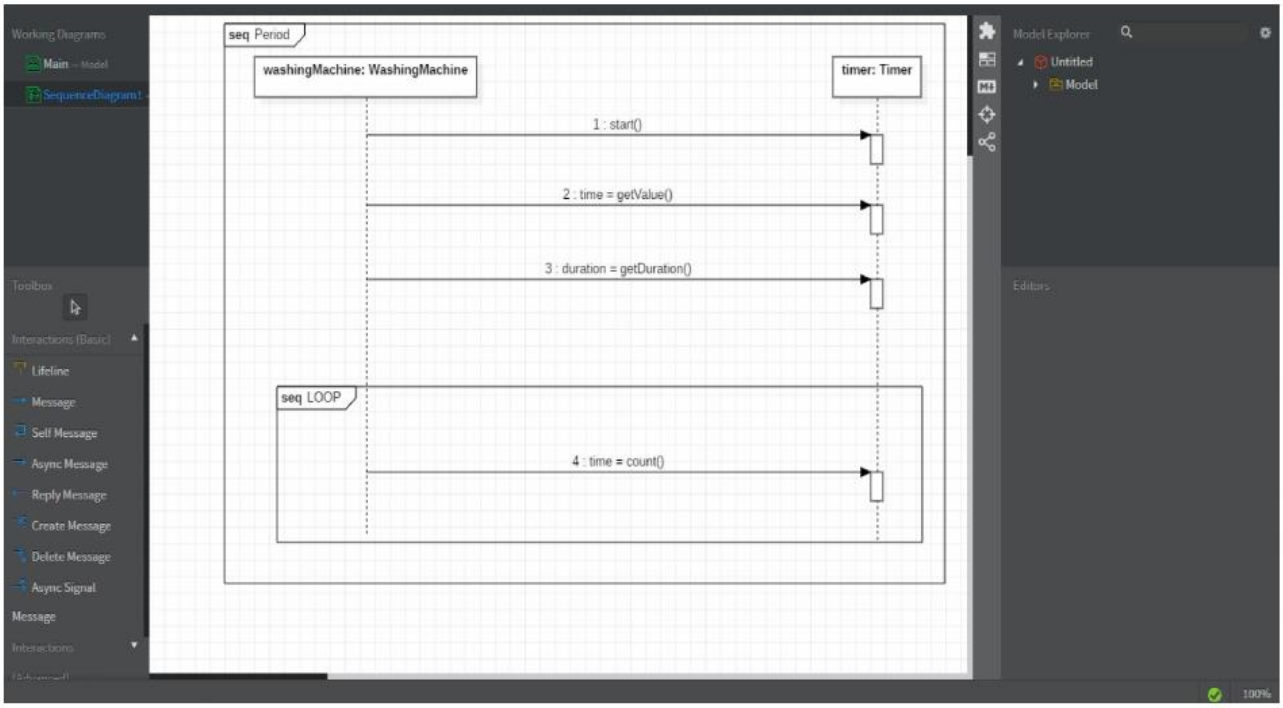
Conclusion :

Learned the importance and use of sequence diagram.


sd Supermarket Automation System
Lifeline1: Admin
Login Success
Store Management
Purchasing Management
Sales Management
Customer Management
Payment Management
1 : Add/Edit Store
2 : Add/Edit Purchasing
3 : Add/Edit Sales
4 : Add/Edit Customer
5 : Add/Edit Payment
6 : Login to page
7 : Manage Store
8 : Save/Update Sales
9 : Save/Update Customer
10 : Save/Update Payment
11 : Save/Update Store
12 : Save/Update Purchasing
13 : Manage Purchasing Details
14 : List/Delete Customer
15 : List/Delete Payment
16 : List/Delete Sales
17 : List/Delete Purchasing
18 : Manage Sales Detail
19 : Manage Customer Details
20 : Manage Payment Details
21 : List/Delete Store


seq SD Main
WashMachine: WashingMachine
washOption: WashOption
2 : option = getWashSelection
1 : Message1
seq alt
[option==1]
3 : StandardWash
[Option==2]
4 : RinseTwice
[Option==3]
5 : Spin
Working Diagrams
Main -- Model
SequenceDiagram1
Toolbox
Interactions (Basic)
Lifeline
Message
Self Message
Async Message
Reply Message
Create Message
Delete Message
Async Signal
Model Explorer
Untitled
Model
Editors
100%


seq Spin
washinhMachine: WasingMachine
engine: Engine
timer: Timer
1 : turnOn()
2 : SetDuration()
seq ref
PERIOD
3 : turnOff()
Working Diagrams
Main -- Model
SequenceDiagram1
Toolbox
Interactions (Basic)
Lifeline
Message
Self Message
Async Message
Reply Message
Create Message
Delete Message
Async Signal
Model Explorer
Untitled
Model
Editors
100%


Working Diagrams
Main – Model
SequenceDiagram1
Toolbox
Interactions (Basic)
Lifeline
Message
Self Message
Async Message
Reply Message
Create Message
Delete Message
Async Signal
Message
Interactions
seq Period
washingMachine: WashingMachine
timer: Timer
1 : start()
2 : time = getValue()
3 : duration = getDuration()
seq LOOP
4 : time = count()
Model Explorer
Untitled
Model
Editors
100%

Code

WaterSensor.java

```
public class WaterSensor {

private int currentLevel;

private int desiredLevel;

private static Sensor sensor;

public WaterSensor() {

sensor = new Sensor();

this.desiredLevel = 500;

this.currentLevel = 0;

}

public WaterSensor(int currentLevel, int desiredLevel) {

sensor = new Sensor();

this.desiredLevel = desiredLevel;

this.currentLevel = currentLevel;

}

public void increaseWaterLevel(int fillAmount) {

this.currentLevel += fillAmount;

}

public int getStatus() {

/*

1: Not Filled

-1: Filled

0: Over fill
```

```
*/
return sensor.check(currentLevel, desiredLevel);
}
public int getCurrentLevel() {
return currentLevel;
}
}
```

WashOption.java

```
public class WashOption {
private int WashSelection;
public int getWashSelection() {
return WashSelection;
}
public void setWashSelection(int option) {
WashSelection = option;
}
}
```

TempSensor.java

```
public class TempSensor {
private boolean status;
public TempSensor() {
status = false;
}
```

```
public boolean getStatus() {
return this.status;
}
public void setStatus(boolean option) {
this.status = option;
}
}
```

Sensor.java

```
public class Sensor {
private DoorSensor doorSensor;
private TempSensor tempSensor;
public Sensor() {
doorSensor = new DoorSensor();
tempSensor = new TempSensor();
}
public int check(int currentLevel, int desiredLevel) {
if (currentLevel < desiredLevel) {
if (tempSensor.getStatus()) {
System.out.println("\nTemperature sensor is On, turning it Off.");
tempSensor.setStatus(false);
}
if (!doorSensor.getStatus()) {
System.out.println("\nWater knob is closed, opening it...");
doorSensor.setStatus(true);
```

```
}
return 1; // underflow
}
else if (currentLevel == desiredLevel) {
if (!tempSensor.getStatus()) {
System.out.println("\nTemperature sensor is On, turning it Off.");
tempSensor.setStatus(true);
}
if (doorSensor.getStatus()) {
System.out.println("\nWater knob is open, closing it...");
doorSensor.setStatus(true);
}
return -1; // filled
}
else {
return 0; // overflow
}
}
}
```

Machine.java

```
public interface Machine {
public void turnoff();
public void turnon();
```

```
}
```

Engine.java

```
public class Engine {
public int rotation;
public Engine() {
rotation = 1000;
}
public Engine(int rotation) {
this.rotation = rotation;
}
}
```

DoorSensor.java

```
public class DoorSensor {
private boolean status;
public DoorSensor() {
status = false;
}
public boolean getStatus() {
return this.status;
}
public void setStatus(boolean option) {
this.status = option;
}
}
```

WashingMachine.java

```
import java.util.Scanner;
public class WashingMachine implements Machine {
private static WaterSensor waterSensor;
private static int washTime;
private static int rinseTime;
// private static int desiredLevel = 500;
public static void main(String[] args) throws InterruptedException {
Machine washingMachine = new WashingMachine();
washingMachine.turnon();
System.out.println("Insert your clothes.");
for (int i = 0; i <= 3; i++) {
Thread.sleep(1000);
System.out.print("\rMachine Waiting for: [" + (3 - i) + "]");
}
System.out.println();
Fill();
Wash();
Rinse();
Empty();
washingMachine.turnoff();

}
```

```
private static void Wash() throws InterruptedException {
System.out.print("Enter wash Duration: ");
washTime = new Scanner(System.in).nextInt();
while (washTime >= 0) {
System.out.print("\rWashing.. (Time Remaining[" + washTime + "])");
Thread.sleep(1000);
washTime--;
}
System.out.println();
}
private static void Fill() throws InterruptedException {
System.out.println("Filling Water");
while (waterSensor.getStatus() == 1) {
waterSensor.increaseWaterLevel(10);
System.out.print("\rCurrent Water Level: " + waterSensor.getCurrentLevel());
Thread.sleep(100);
}
}
private static void Empty() throws InterruptedException {
System.out.println("Draining");
Thread.sleep(1000);
System.out.println("Clothes drained.");
System.out.println("Process Completed.\n");
}
```

```
private static void Rinse() throws InterruptedException {
System.out.print("Enter Rinse Time: ");
rinseTime = new Scanner(System.in).nextInt();
System.out.println("Spinning Clothes");
while(rinseTime >= 0) {
System.out.print("\rTime Remaining [" + rinseTime + "]");
rinseTime--;
Thread.sleep(1000);
}
System.out.println("\n");
}
@Override
public void turnoff() {
System.out.println("Machine Turned Off\n");
}
}
```

Output

```
Machine Turned On
Insert your clothes.
Machine Waiting for: [0]
Filling Water
Water knob is closed, opening it...
Current Water Level: 500
```

Temperature sensor is On, turning it Off.

Water knob is open, closing it...

Enter wash Duration: 5

Washing.. (Time Remaining[0])

Enter Rinse Time: 4

Spinning Clothes

Time Remaining [0]

Draining

Clothes drained.

Process Completed.

Machine Turned Off

Assignment 7

Aim – To prepare state diagram

Objective – To learn behavioral modeling & create state diagram

Theory – It describes different states of a components in a system these states are specific to a component object of a system. A state chart diagram is one of the five UML diagram used its model dynamic nature of a system. They define different states of an object during its lifetime & these states are changed by events so state chart diagram are useful to model reactive systems.

Main purpose of using state chart diagram

- To model dynamic aspect of a system
- To model lifetime of a reactive systems
- To describe different states of an object during the lifetime
- Define a state machine to model states of an object

How to draw state chart diagram?

Before drawing classify the following points

- Identify important objects be analysed
- Identify the states
- Identify the events

Notations in the state diagrams:

Events:

It is something that happens that affects the systems. Eg. Keystroke is an event for keyboard

States:

A state captures the relevant aspect of the system history very effectively. Eg. When you press a key the character generated will depend whether CAPSLOCK is ON or NOT

Ground Conditions:

These are Boolean expansions evaluated dynamically based on the value of the extended state valuable & event parameters

Actions & Transitions :

When an event instaces is displayed the state machine responds by performing actions such as changing a variable performing I/O, invoking a function, generating another want instance or changing to another state

Where to use state diagram?

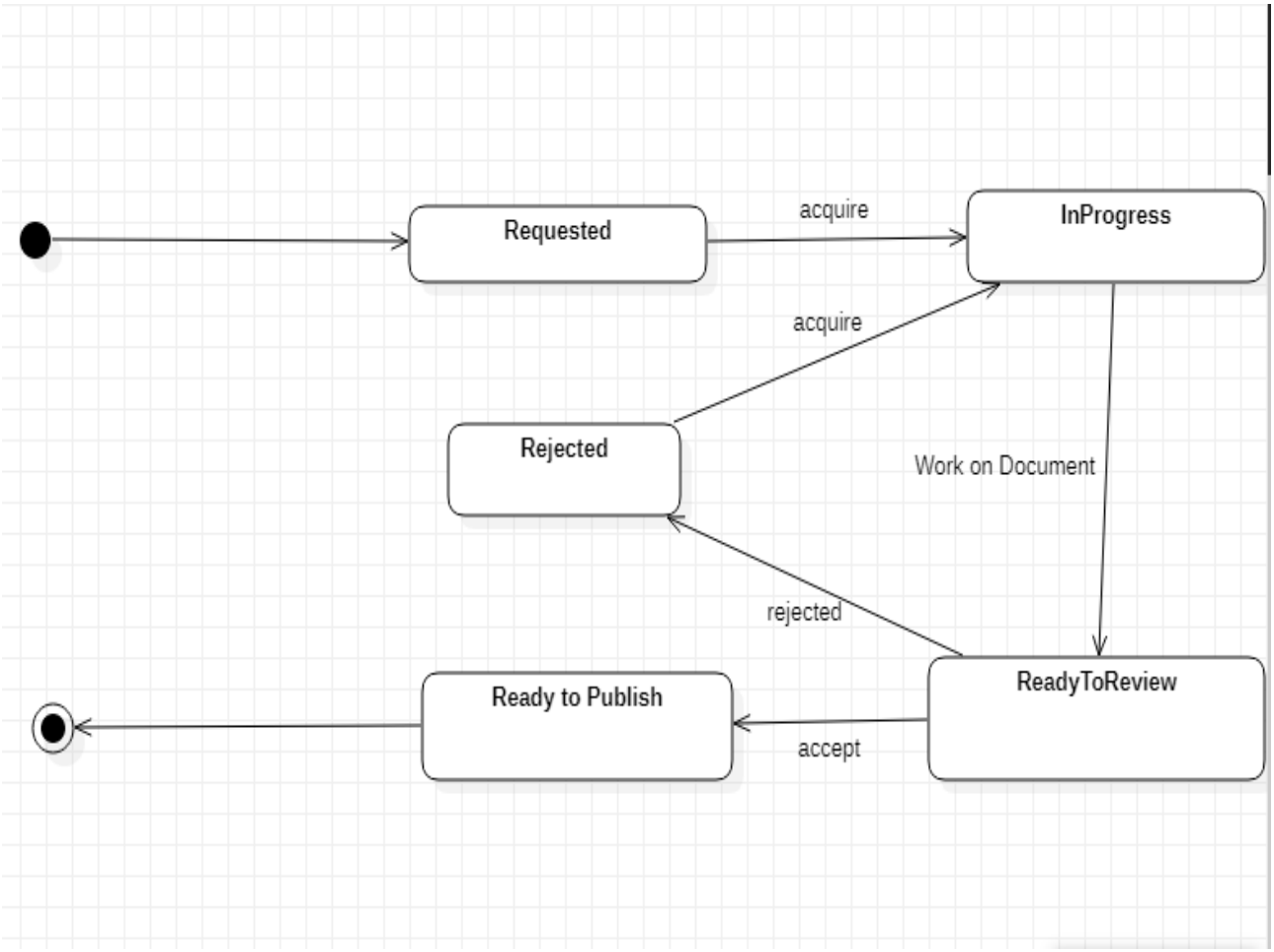
They are used to model dynamic aspects of a system like other four diagrams they are used to model state & also event operating on the system.

So,the main usage can be described as

- To model objects state of the system
- To model reactive system
- To identify event responsible for state change
- Forward & reverse engineering

Conclusion:

Importance & use of state model is studied. Drawn the state diagram using the notation & implemented using OO language.


Requested
acquire
InProgress
acquire
Rejected
Work on Document
rejected
ReadyToReview
Ready to Publish
accept

State.java

```
public interface State (
public void acquire(Context context);
public void rejected(Context context);
public void work(Context context);
public void accept(Context context);
}
```

Requested.java

```
public class Requested implements State (
        Requested(){
                System.out.println("State Changed to Requested");
        }
        public void acquire(Context context) {
                context.setState(new In Progress());
        }
        public void rejected(Context context) {}
        public void work(Context context){}
        public void accept(Context context) {}
)
```

Rejected.java

```
public class Rejected implements State{
```

```
        Rejected(){
                System.out.println("State changed to Rejected");
        }
        public void acquire(Context context) {
                context.setState(new InProgress();
        }
        public void rejected(Context context){}
        public void work(Context context) {}
        public void accept(Context context) {}
}
```

Ready To Review.java

```
public class ReadyToReview implements State (
        Ready To Review(){
                System.out.println("State changed to Ready To Review');
        }
        public void acquire(Context context) {}
        public void rejected(Context context) {
                context setState(new Rejected());
        }

        public void work(Context context) {}
        public vold accept(Context context) {
                context setState(new Ready To Publish();
```

```
        }
```

ReadyToPublish.java

```
public class Ready ToPublish implements State{
        ReadyToPublish() {
                System.out.println("State Changed to Ready To Publish");
        }
        public void acquire(Context context) {}
        public void rejected(Context context) {}
        public void work(Context context){}
        public void accept(Context context){}
}
```

In Progress.java

```
public class In Progress implements State {
        InProgress(){
        System.out.println("State changed to InProgress");
        }
        public void rejected(Context context) {}
        public void acquire(Context context) {}
        public void work(Context context) {
                context.setState(new ReadyToReview());
        }
        public void accept(Context context) {}
```

}

Context.java

```
public class Context{
                    private State state;
                    Context() {
                            this.state-new Requested();
                    }
                    public void setState(State state) {
                            this.state = state;
                    }
                    public void acquire() {
                             state.acquire(this);
                    }
                    public void rejected() {
                            state rejected(this);
                    }
                    public void work() {
                            state.work(this);
                    }
                    public void accept() {
                            state.accept(this);
                    }
}
```

Client.java

```
public class client{
        public static void main(String] args) {
            Context document new Context();
                document.acquire();
                document.work();
                document.rejected();
                document.acquire();
                document.work();
                document.accept();
        }
}
```

Output

State Changed to Requested

State changed to In Progress

State changed to Ready To Review

State changed to Rejected

State changed to In Progress

State changed to Ready To Review

State Changed to Ready To Publish

ASSIGNMENT NO: 8

AIM:

Identification and Implementation of GRASP patterns.

DESCRIPTION:

Apply any two GRASP patterns to define the design model for given problem description using effective UML 2 design and implement them with a suitable object-oriented language.

THEORY:

GRASP consist of guidelines for assignment responsibility to classes and objects in object-oriented design. The different pattern and principles used in GRASP are controller, creator, indirection, information experts, high cohesion, low coupling, polymorphism, protected variation and pure factorisation.

Responsibilities are more general than methods. Methods are implemented to fulfil responsibilities.

GRASP Principles:

1. Controller
2. Creator
3. Indirection
4. Information experts
5. High cohesion
6. Low coupling
7. Polymorphism
8. Protected variation
9. Pure factorisation.

GRASP Patterns:

The GRASP patterns are a learning aid to help one understand essential object design and apply design principles are based on pattern. Design reasoning is a methodical, rational pattern. The approach to understand and using design principles are based on pattern of assigning responsibilities.

Responsibilities and Methods:

The UML defines a responsibility as contact or obligation of a classifies responsibilities are its behaviour. Basically, responsibilities are of the following two types:

• Knowing

• Doing

POS application are used to explain all the GRASP patterns:

1. Application for shop, restaurant etc that register sale.
2. A product has a specification including a description unitary price and identifier.
3. The application also register payment associated with sales.
4. A payment is for a certain amount, equal or greater than the total of the sale.

Low Coupling GRASP pattern:

Problem: How to support low dependency, low change impact and incredible reuse?

Solution: Assign responsibilities so that coupling remain. Try to avoid one class to have to know about other things.

Controller: Assign the responsibility for receiving or handling a system event message to a class representing one of the following choices:

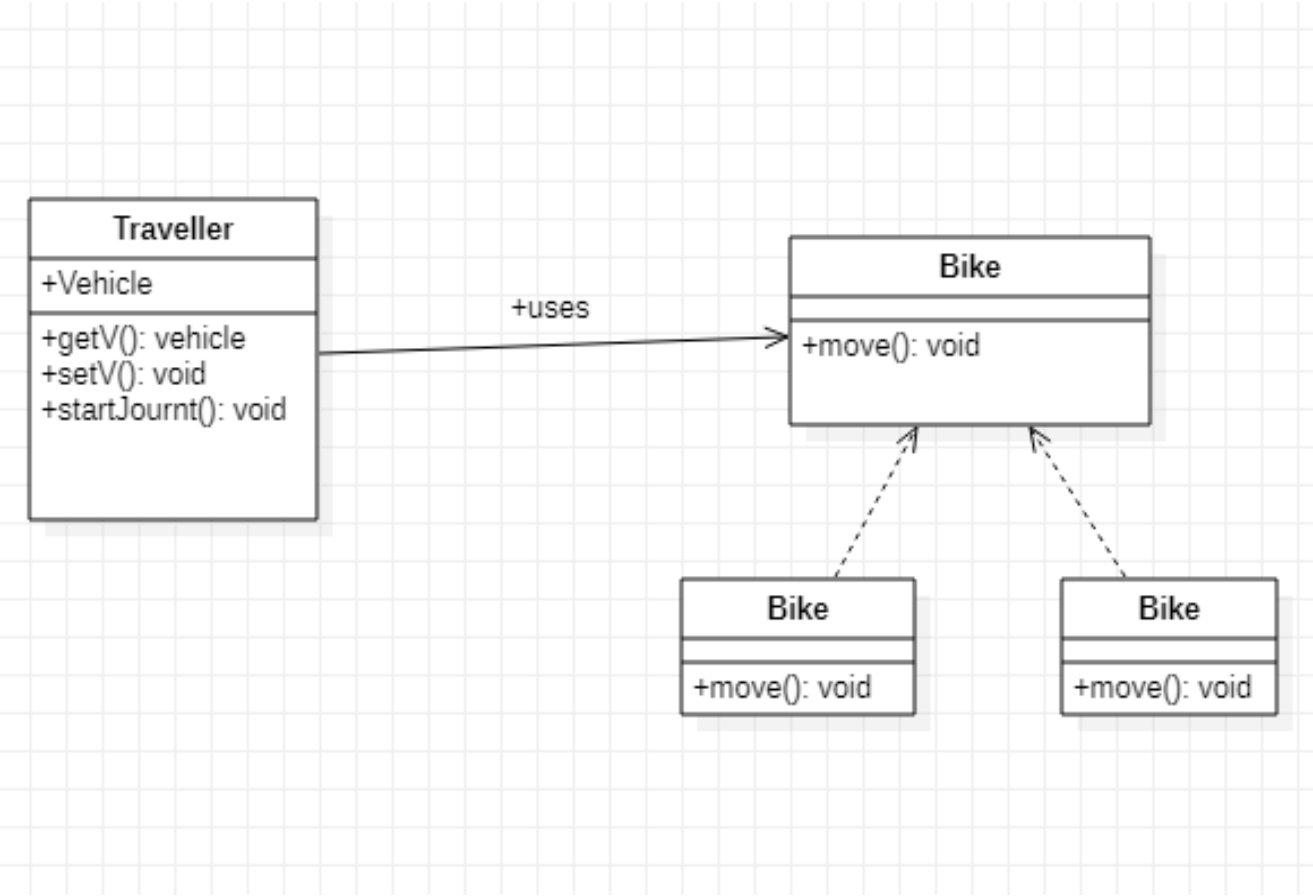
- § Represents the overall system device and subsystems.
- § Represents a use case scenario.
- § Use the same controller class for all systems.

Creator GRASP Pattern: Assign class B the responsibility to create an instance of class A or move of the following is true:

- Ø B aggregates A objects.
- Ø B contains A objects.
- Ø B record instance of A object
- Ø B closely use A object.

CONCLUSION:

GRASP consists of guidelines for assigning the responsibilities to classes and object in object-oriented design are studied.


Traveller
+Vehicle
+getV(): vehicle
+setV(): void
+startJournt(): void
+uses
Bike
+move(): void
Bike
+move(): void
Bike
+move(): void

Bike.java

```
Class Bike implements Vehicle{
        @Override public Void move()
        {
                System.out.println(”Bike is moving");
        }
}
```

Car.Java

```
Class Car implements Vehicle{
        @Override public void move()
        {
                System.out.println(“Car is moving“);
        }
}
```

Vehicle.java

```
public interface Vehicle
{
        publiC VOid move();
}
```

Traveler.java

```
class Traveler
{
```

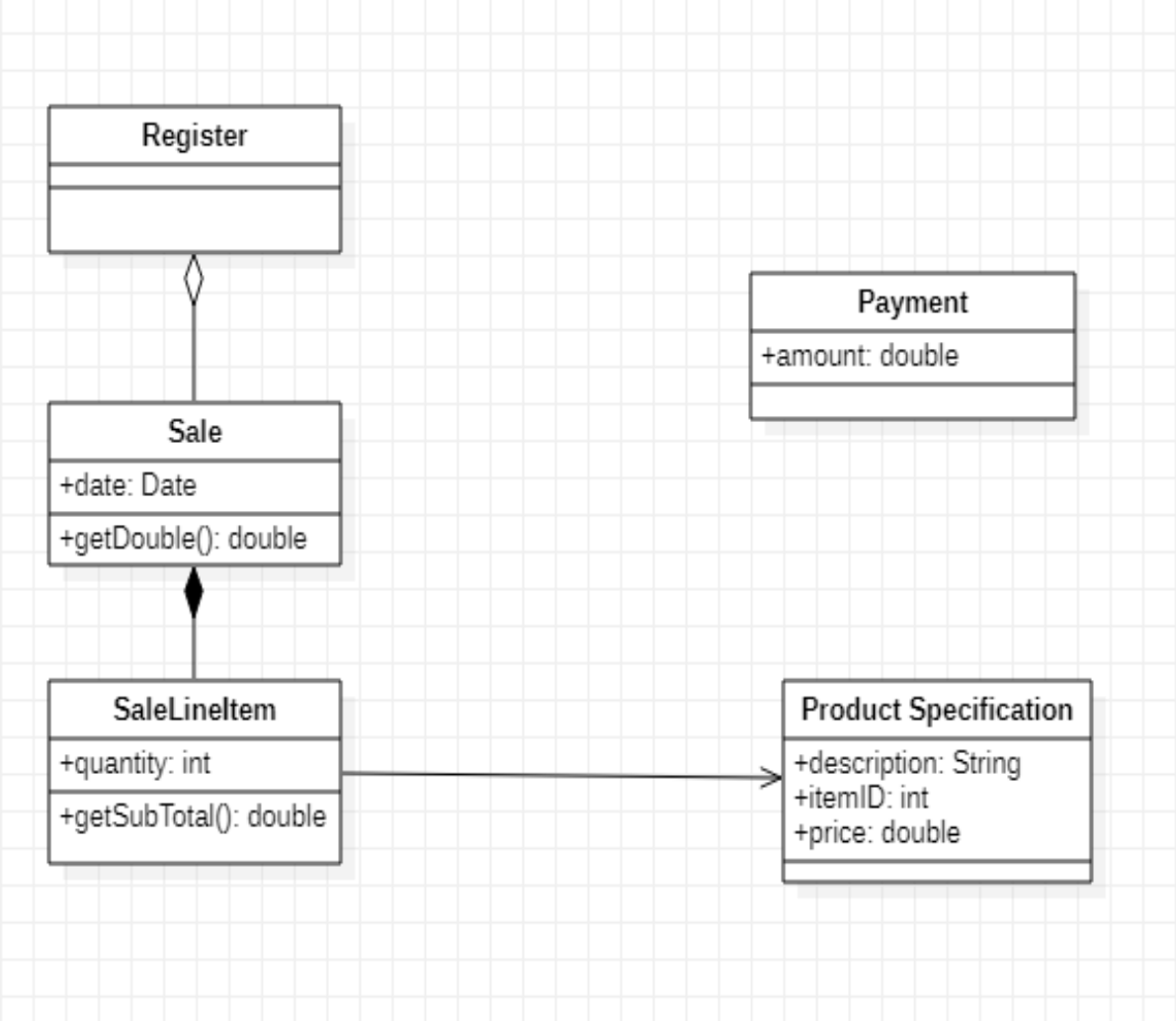
```
        private Vehicle v;
        public Vehicle getV()
        { return v; }
        public void setV(Vehicle v)
        { this.v= v; }
        public void starUourney()
        { v.move(); }
}
```

Test.java

```
public class Test
{
        public static void main(String[] args)
        {
                Traveler traveler= new Traveler();
                traveler.setV(new Car());  // inject car dependency
                traveler.starUoumey(); //startjourney by car
                traveler.setV(new Bike()); // inject bike dependency
                traveler.startJourney(); //startjourney by bike
        }
}
```

Output

```
car is moving
Bike is moving
```


Register
Payment
+amount: double
Sale
+date: Date
+getDouble(): double
SaleLineItem
+quantity: int
+getSubTotal(): double
Product Specification
+description: String
+itemID: int
+price: double

Sale class

```
Public class Gate(){
        //. .
        public double getTotal(){
                double total=0;
                for(SaleLineItems : saleaLineItem){
                        ProductSpecification prodspec= s.getProductSpecification();
                        total+= s.getQuantity()*prodspec.getPrice();
                }
                return total;
        }
}
```

Subtotal class

```
public class SaleLineItem{
        //..
        public double getSubTotal(){
                productSpecification.getPrice();
                productSpecification.getPrice();
        }
}
```

Main Class

```
public Class {
        public static void main(String args[]){
                System.out.println("Calculating Subtotal….\n");
                SaleLineItem.getSubTotal();
                Thread. sleep(1000);
                int total_price: Sale.getTotal();
                System.out.println("The total Price for Payment is:");
                System.out.println (total price);
        }
}
```

OUTPUT:

calculating SubtOtal....

The total Price for Payment is:

1200

Assignment 9

Aim:

Identification and Implementation of GOF pattern

Description

:
Apply any two GOF pattern to refine Design Model for a given problem description Using effective UML 2 diagrams and implement them with a suitable object oriented language

Theory

Design patterns represent the best practices used by experienced object-oriented software developers. Design patterns are solutions to general problems that software developers faced during software development. These solutions were obtained by trial and error by numerous software developers over quite a substantial period of time.

Common platform for developers:

Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

Best Practices Design patterns

They have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps unexperienced developers to learn software design in an easy and faster way.

Uses of Design Patterns

Design patterns can speed up the development process by providing tested, proven development
paradigms. Effective software design requires considering issues that may not become visible

until later in the implementation. Reusing design patterns helps to prevent subtle issues that can
cause major problems and improves code readability for coders and architects familiar with the patterns.Often, people only understand how to apply certain software design techniques to certain problems

Creational design patterns

This design patterns is all about class instantiation. This pattern can be further divided into classcreation patterns and object-creational patterns. While class-creation patterns use inheritance effectively in the instantiation process, object-creation patterns use delegation effectively to get the job done.

AbstractFactory:

Creates an instance of several families of classes
• Builder
Separates object construction from its representation
• FactoryMethod
Creates an instance of several derived classes
• ObjectPool
Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
• Prototype
A fully initialized instance to be copied or cloned
• Singleton
A class of which only a single instance can exist

Structural design patterns:

This design patterns is all about Class and Object composition. Structural class-creation patterns
use inheritance to compose interfaces. Structural object-patterns define ways to compose objects
to obtain new functionality.
• Adapter
Match interfaces of different classes
• Bridge
Separates an object’s interface from its implementation
• Composite

A tree structure of simple and composite objects
• Decorator
Add responsibilities to objects dynamically
• Facade
A single class that represents an entire subsystem
• Flyweight
A fine-grained instance used for efficient sharing
• PrivateClassData
Restricts accessor/mutator access
• Proxy
An object representing another object

Behavioral design patterns:

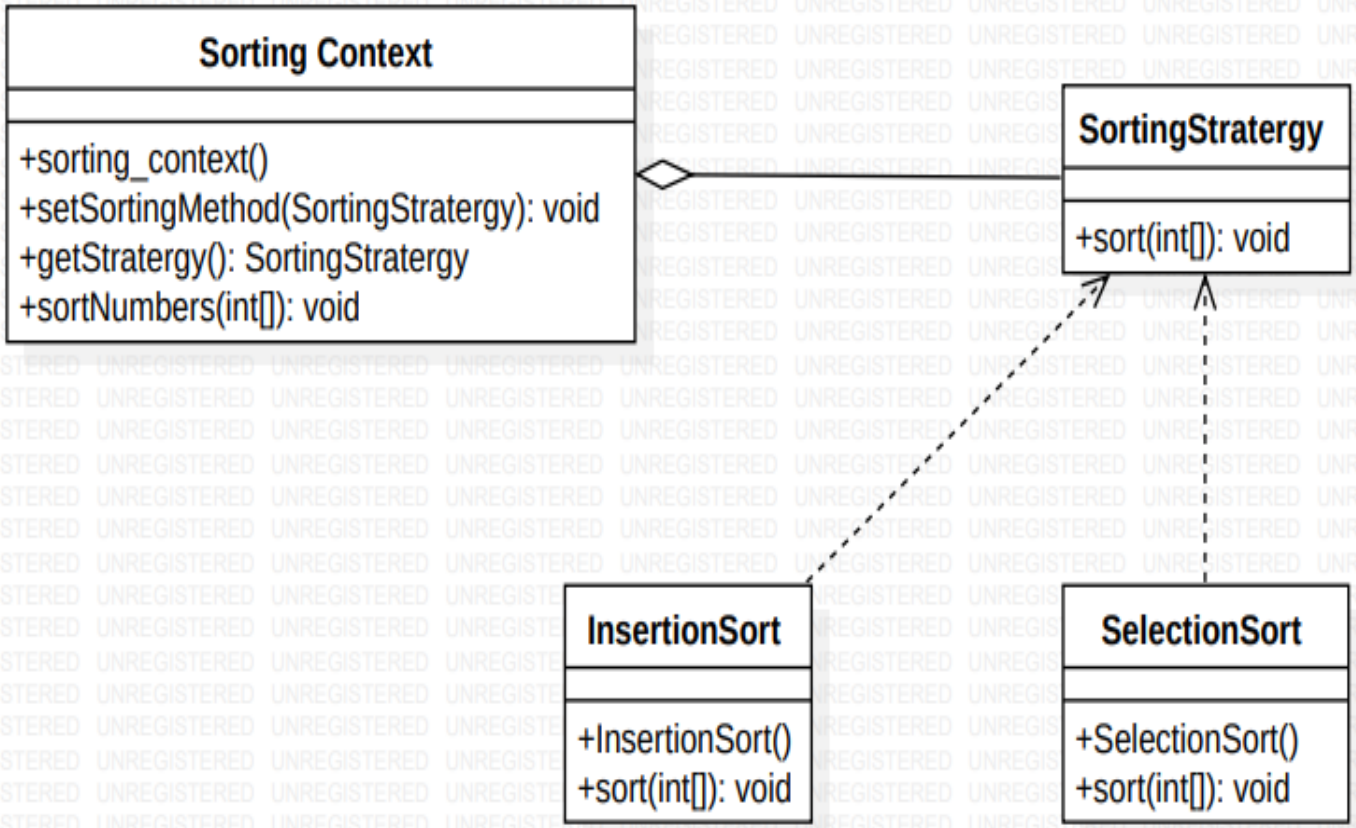
This design patterns is all about Class's objects communication. Behavioral patterns are those
patterns that are most specifically concerned with communication between objects.
• Chainofresponsibility
A way of passing a request between a chain of objects
• Command
Encapsulate a command request as an object
• Interpreter
A way to include language elements in a program
• Iterator
Sequentially access the elements of a collection
• Mediator
Defines simplified communication between classes
• Memento
Capture and restore an object's internal state
• NullObject
Designed to act as a default value of an object
• Observer
A way of notifying change to a number of classes

Context:

o maintains a reference to the current Strategy object
o supports interface to allow clients to request Strategy calculations
o allows clients to change Strategy

Conclusion:

GOP design pattern studied and implemented.


Sorting Context
+sorting_context()
+setSortingMethod(SortingStratergy): void
+getStratergy(): SortingStratergy
+sortNumbers(int[]): void
SortingStratergy
+sort(int[]): void
InsertionSort
+InsertionSort()
+sort(int[]): void
SelectionSort
+SelectionSort()
+sort(int[]): void

SortingStrategy.java

```
public interface SortingStrategy {
public void sort(int[] numbers);
}
```

SelectionSort.java

```
public class SelectionSort implements SortingStrategy {
@Override
public void sort(int[] numbers) {
System.out.println("Selection Sort!");
int i, j, first, temp;
for (i = numbers.length - 1; i > 0; i--) {
first = 0;
for (j = 1; j <= i; j++) {
if (numbers[j] > numbers[first])
first = j;
}
temp = numbers[first];
numbers[first] = numbers[i];
numbers[i] = temp;
}
System.out.println(Arrays.toString(numbers));
}
}
```

InsertionSort.java

```
public class InsertionSort implements SortingStrategy {
@Override
public void sort(int[] numbers) {
System.out.println("Insertion Sort!");
for (int i = 1; i < numbers.length; i++) {
int temp = numbers[i];
int j;
for (j = i - 1; (j >= 0) && (numbers[j] > temp); j--) {
numbers[j + 1] = numbers[j];
}
```

```
numbers[j + 1] = temp;
}
System.out.println(Arrays.toString(numbers));
}
}
public class SortingContext {
private SortingStrategy strategy;
public void setSortingMethod(SortingStrategy strategy) {
this.strategy = strategy;
}
public SortingStrategy getStrategy() {
return strategy;
}
public void sortNumbers(int[] numbers){
strategy.sort(numbers);
}
}
```

TestMain.java

Here is how client using strategy pattern

```
public class TestMain {
public static void main(String[] args) {
int numbers[] = {20, 50, 15, 6, 80};
SortingContext context = new SortingContext();
context.setSortingMethod(new InsertionSort());
context.sortNumbers(numbers);
System.out.println("**********");
context.setSortingMethod(new SelectionSort());
context.sortNumbers(numbers);
}
}
```

Output

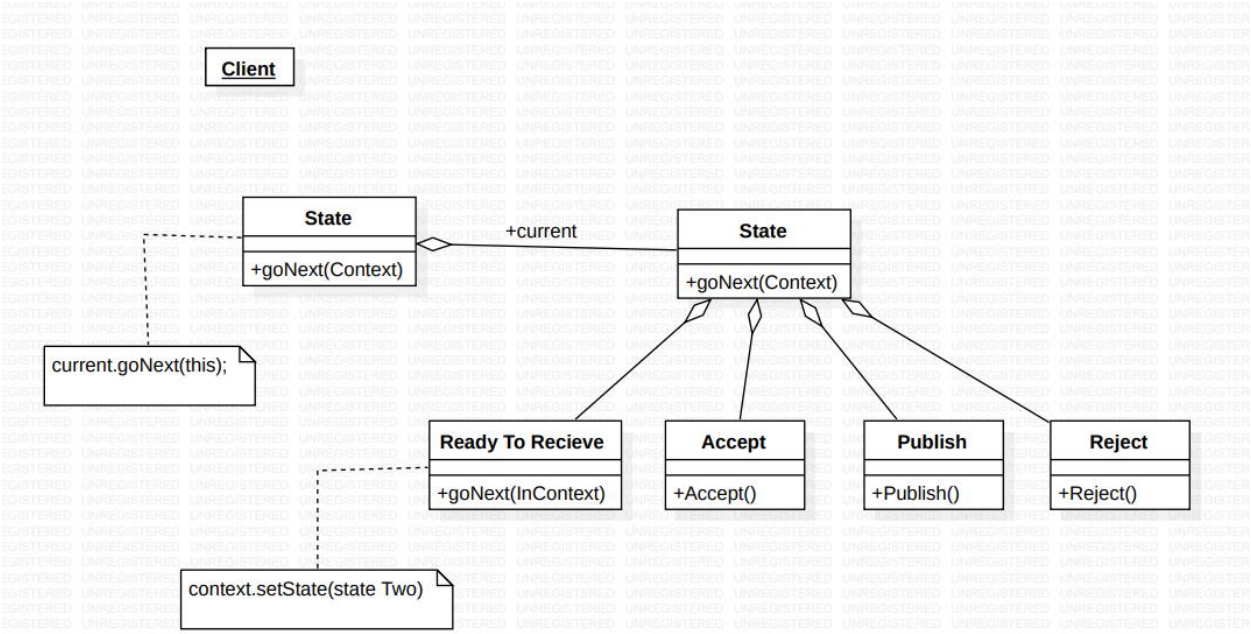
Insertion Sort!

[6, 15, 20, 50, 80]

****

Selection Sort!

[6, 15, 20, 50, 80]



State.java

```
public interface State {
public void acquire(Context context);
public void rejected(Context context);
public void work(Context context);
public void accept(Context context);
}
```

Requested.java

```
public class Requested implements State {
Requested() {
System.out.println("State Changed to Requested");
}
public void acquire(Context context) {
context.setState(new InProgress());
}
public void rejected(Context context) {}
public void work(Context context) {}
public void accept(Context context) {}
}
```

Rejected.java

```
public class Rejected implements State {
Rejected(){
System.out.println("State changed to Rejected");
}
public void acquire(Context context) {
context.setState(new InProgress());
}
public void rejected(Context context) {}
public void work(Context context) {}
public void accept(Context context) {}
}
```

ReadyToReview.java

```
public class ReadyToReview implements State {
ReadyToReview(){
```

```
System.out.println("State changed to Ready To Review");
}
public void acquire(Context context) {}
public void rejected(Context context) {
context.setState(new Rejected());
}
public void work(Context context) {}
public void accept(Context context) {
context.setState(new ReadyToPublish());
}
}
```

ReadyToPublish.java

```
public class ReadyToPublish implements State {
ReadyToPublish() {
System.out.println("State Changed to ReadyToPublish");
}
public void acquire(Context context) {}
public void rejected(Context context) {}
public void work(Context context) {}
public void accept(Context context) {}
}
```

InProgress.java

```
public class InProgress implements State {
InProgress() {
System.out.println("State changed to InProgress");
}
public void acquire(Context context) {}
public void rejected(Context context) {}
public void work(Context context) {
context.setState(new ReadyToReview());
}
public void accept(Context context) {}
}
```

Context.java

```
public class Context {
private State state ;
Context() {
this.state=new Requested();
}
public void setState(State state) {
this.state = state;
}
public void acquire() {
state.acquire(this);
}
public void rejected() {
state.rejected(this);
}
public void work() {
state.work(this);
}
public void accept() {
state.accept(this);
}
}
```

Client.java

```
public class client {
public static void main(String[] args) {
Context document = new Context();
document.acquire();
document.work();
document.rejected();
document.acquire();
document.work();
document.accept();
}
```

```
}
```

Output

```
State Changed to Requested
State changed to InProgress
State changed to Ready To Review
State changed to Rejected
State changed to InProgress
State changed to Ready To Review
State Changed to ReadyToPublish
```