PAGE No.: 2
DATE :

Practical No. 1.

Aim: Write a program to demonstrate following OOP Principle
1. "Program to an interface, not an implementation".
2. "Encapsulate the aspects of the system that varies".
3. "Favour composition over inheritance"

INLAB

Aim: Write a program to create a duck simulator game.

Objectives:
- To understand the need of programming to an interface over inheritance.
- To identify the aspects of application that vary and separates from others.
- To identify the requirement of favouring composition over inheritance.

Theory: The design principles are the set of advice used as aids in design making. The design principles are similar to the design patterns concepts. The only difference between the design principle and design pattern is that the design principles are more generalized and abstract. The design pattern contains much more practical advice and concrete. The design patterns are related to the entire class problems, not just generalized coding practices.

There are two design principle :-

(1) Program to an interface rather than implementation

(2) Identify the aspect that vary and separate them from what stays the same.

(1) Program to an interface rather than implementations states that say such modules should have an abstract supertype like an interface. The basic methods should be made available in the interface.

(2) Strategy pattern :- strategy patterns comes under the Behavioral pattern. If you have number of algorithm to perform any task or function then seperate that function as interface, then create different classes which will be implementing the specific technique the interface.

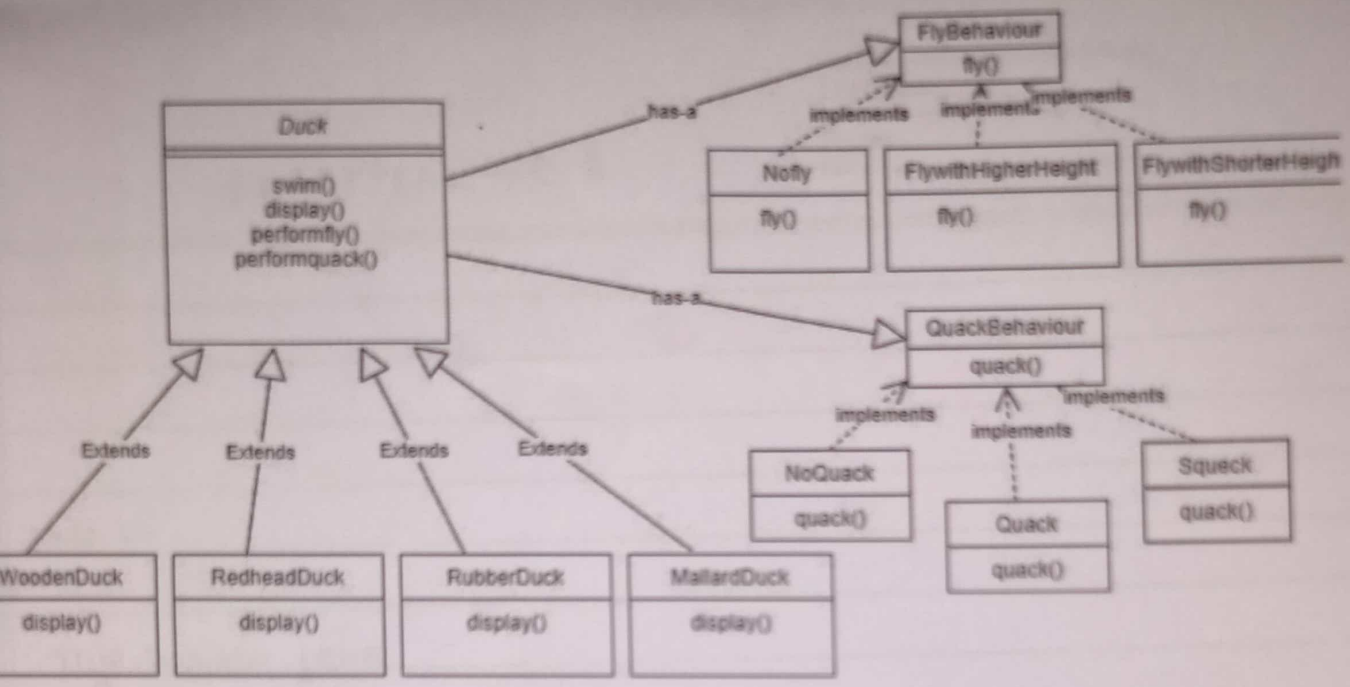
All ducks have some shared properties But some ducks can fly and some (like wooden and Rubber ducks) cannot, and some ducks can quack and some cannot like Squeck and mute ducks).

We know that fly() and quack() are parts of Duck class. To separate these behaviors from the Duck class, pull these methods out of the Duck class and create a new set of classes to represent each behavior. The Behaviors are fly behavior and Quack Behavior. These Behaviors can become interfaces because they are the variable qualities. Fly behavior interface will

have a fly() method. QuackBehaviour interface will have a quack() method.

The Duck class becomes an abstract class and declares 2 reference variables, fb and qb. The Duck abstract also delegates fly and quack behaviour using these.
We can implement the interface like so: new class NoFly, FlywithLongerHeight, FlywithShorterHeight, implements FlyBehaviour. We define the fly() method there as FlyBird, no fly, etc. And new class Quack(), Squeck() and NO quack, implements QuackBehaviour. We define the quack method to make quack and Squeck and No sound.

Any New ducks create will extend the Duck class. Define new class like ~~Mallard~~ MallardDuck, RedHeadDuck, etc. extend Duck and it uses both interfaces and have the display(). The create DuckTest class to test everything with a main() and we create instance of all the Ducks call the methods.


Duck
swim()
display()
performfly()
performquack()
has-a
FlyBehaviour
fly()
implements
implements
implements
Nofly
fly()
FlywithHigherHeight
fly()
FlywithShorterHeigh
fly()
has-a
QuackBehaviour
quack()
implements
implements
implements
NoQuack
quack()
Quack
quack()
Squeck
quack()
Extends
Extends
Extends
Extends
WoodenDuck
display()
RedheadDuck
display()
RubberDuck
display()
MallardDuck
display()

PAGE No.: 13
DATE :

INLAB

Aim :- Write a programm to create a pizza store that has Franchisis in two different parts of the country adding their own special touchup to the pizza, separate the process of creating a pizza with the process of preparing/ ordering a pizza using factory method.

Objectives :- • To understand the importance of creating abstract classes
• To identify the need of separating the process of object creations from rest of the system.

Theory :- The factory method pattern defines an interface for creating an object, but lets subclasses decide which class to instantiate. Abstract creator gives us an interface with a method for creating objects, also known as "factory method". Any other method implemented in the abstract creator are written to operate on products produced by the factory method. Only subclasses actually implement the factory method and create products.

To implement factory method pattern, we have take an example of pizza as a product. As we have only one product, we will be using factory method pattern to solve this problem. We have created pizza class as abstract class & declared one abstract method recipe() & remaining concrete methods as bake(), cuts() & pack() and ingredients().

PAGE No.:

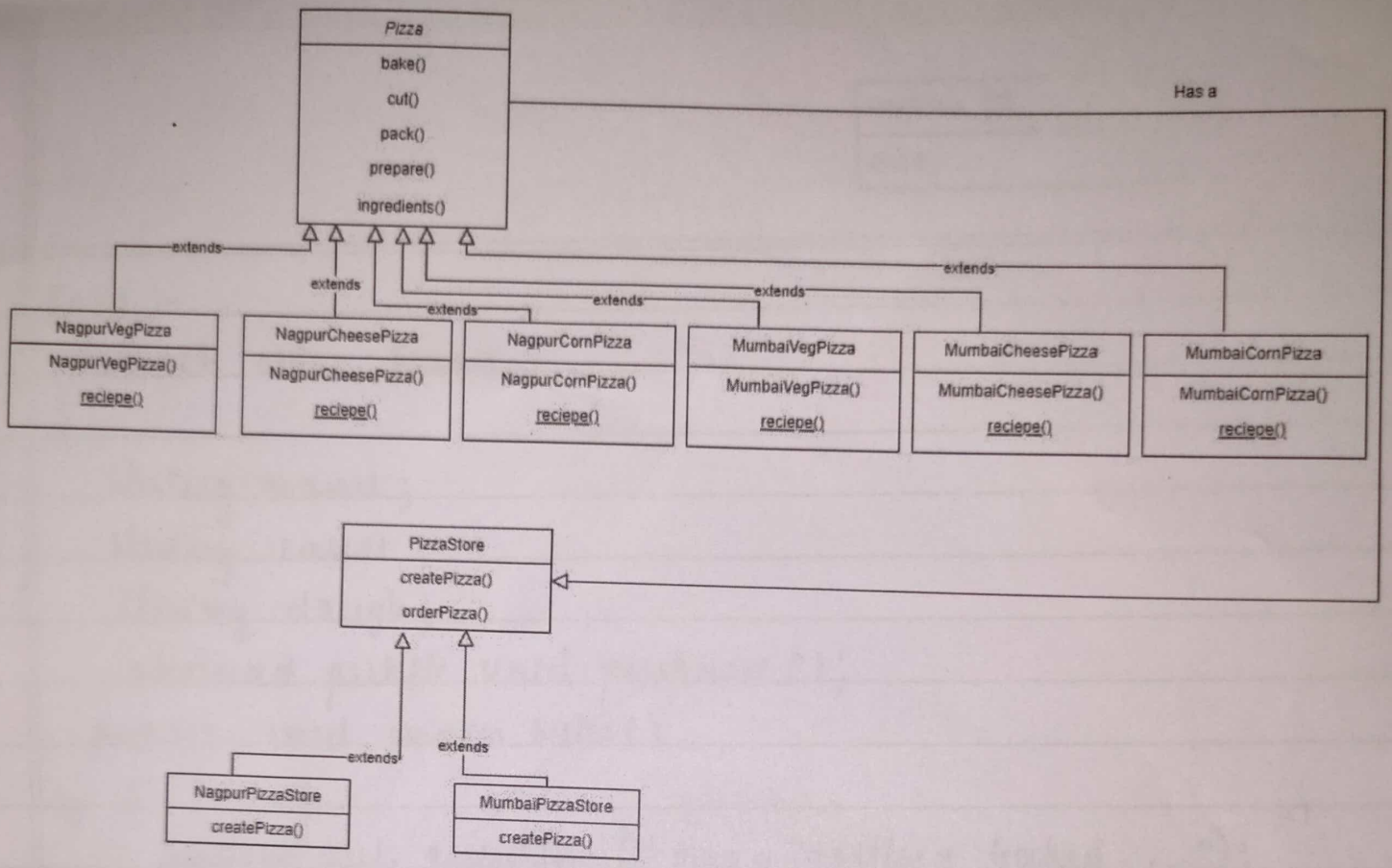
DATE :

Later on, we have created subclasses of pizza class like NagpurCheesePizza, NagpurPaneerPizza, NagpurVegPizza, MumbaiCheesePizza, MumbaiPaneerPizza, & MumbaiVegPizza we have initialized names of an ingredients in the constructor of each subclass and added their own concrete methods like lotsofcheese(), ~~lots of pc~~ lotsofpaneer() etc., Also define receipe() method in each subclass.

Then created a Pizzastore class as an abstract to separate the process of creating a pizza from ordering a pizza. In pizzastore, we have created a reference of Pizza class & orderpizza (String name) method passing name (name of pizza) as an argument & called methods like createpizza(name), prepare(), ingredient(), bake(), cut() & pack(). Included one abstract method createpizza (string name).

Created subclasses NagpurPizzaStore and MumbaiPizzastore extended from PizzaStore. Defined method createPizza in it by initializing a reference of pizza class as "Null" & created varieties of pizza according to the conditions in if-else ladder in both the subclasses & returned a reference value.

To test & begin the execution, created a PizzaDemo class having main() method in it. Created six references of Pizzastore class. & then created objects of NagpurPizzaStore() & MumbaiPizzastore() classes & called orderpizza() method by passing name of pizza as an argument in each step of object creation.


Pizza
bake()
cut()
pack()
prepare()
ingredients()
Has a
extends
extends
extends
extends
extends
extends
NagpurVegPizza
NagpurVegPizza()
reciepe()
NagpurCheesePizza
NagpurCheesePizza()
reciepe()
NagpurCornPizza
NagpurCornPizza()
reciepe()
MumbaiVegPizza
MumbaiVegPizza()
reciepe()
MumbaiCheesePizza
MumbaiCheesePizza()
reciepe()
MumbaiCornPizza
MumbaiCornPizza()
reciepe()
PizzaStore
createPizza()
orderPizza()
extends
extends
NagpurPizzaStore
createPizza()
MumbaiPizzaStore
createPizza()

PAGE No.: 29
DATE :

Practical - 3

Aim: Write a program to demonstrate Abstract factory design pattern

In Lab

Aim: Write a program to create a pizza store that has franchises in different parts of the country using their own set of ingredients to the pizza. Use abstract factory method pattern to provide the means of creating a family of products (ingredients of pizzas) by separating the process of preparing/ ordering a pizza.

Objectives:- • To identify the need of separating the process of object creations from rest of the system.
• To provide an interface for creating families of objects.

Theory:- Abstract Factory is a creational design pattern that lets you produce families of related objects without specifying their concrete classes. It defines an interface for creating all distinct products but leaves the actual product creation to concrete factory classes.

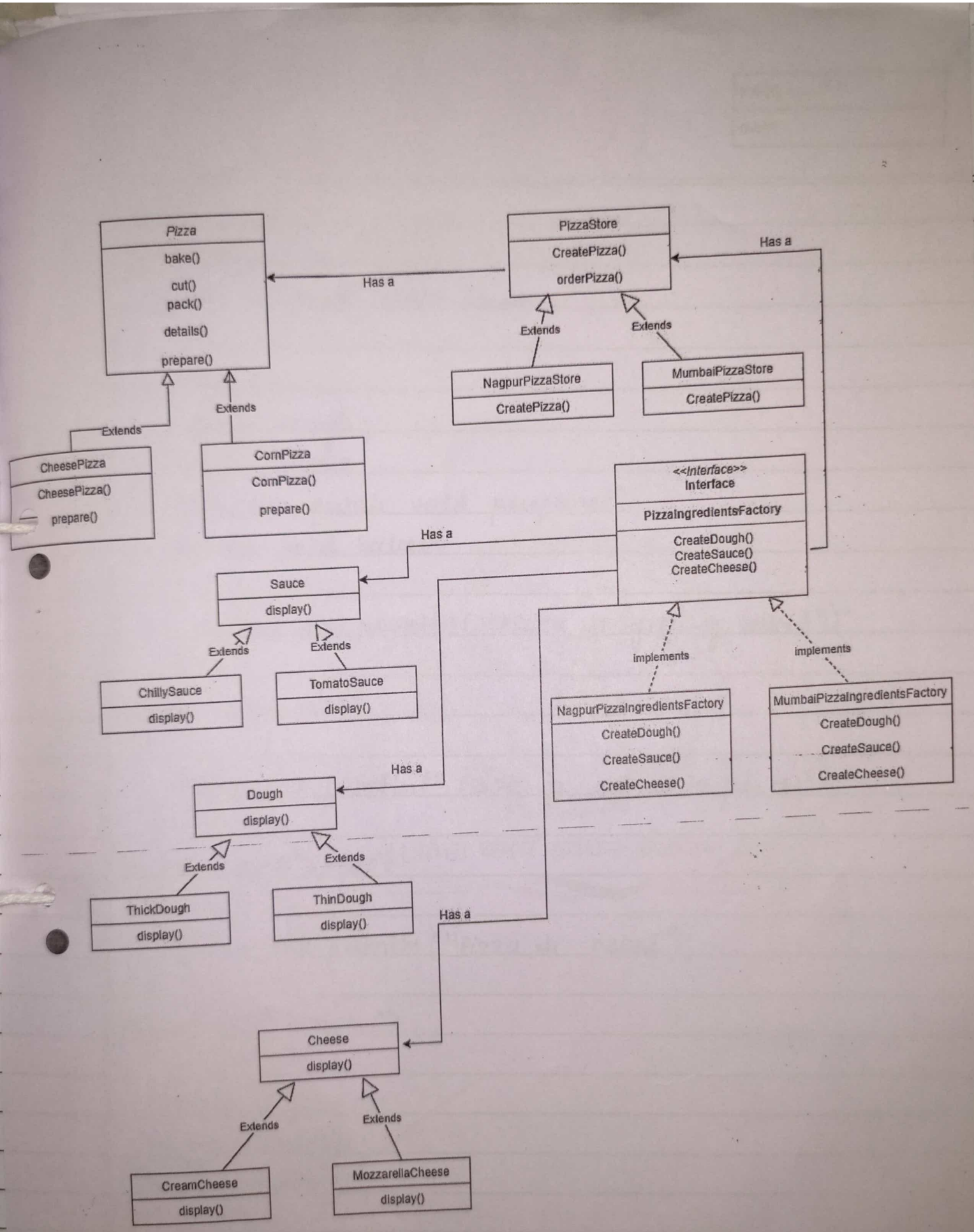
We have created a pizzastore application to demonstrate the implementation of the abstract factory pattern. First created an abstract class Pizza which abstract pizza-related data. Created concrete pizza classes "CheesePizza" & "ClamPizza" which extends Pizza.

Similarly, created abstract classes of product Dough, Sauce & Cheese with their subclasses like ThinDough, ThickDough, TomatoSauce, ChilliSauce, MozzarellaCheese & MozzarellaCheese. Each subclasses defines its display() method.

Created PizzaIngredientFactory to create Dough, Sauce & Cheese. NagpurPizzaIngredientsFactory & MumbaiPizzaIngredients Factory extends from PizzaIngredientFactory. It defines the createDough(), createSauce() & createCheese() methods.

Then created PizzaStore which contains the method orderPizza() in which we createPizza by calling prepare(), details(), bake(), cut() & pack() methods. Then created NagpurPizza store class extends from PizzaStore which produces Pizza object based on the type of the Pizza. Similarly, created MumbaiPizzaStore which also produces pizza object based on type such as CheesePizza or CrunchPizza.

To test the Abstract Factory pattern, we have created PizzaDemo class in which we have four different objects created varies from NagpurPizzaStore's pizza to MumbaiPizzaStore's Pizza & got the type of pizza we have passed in orderPizza() method.


Pizza
bake()
cut()
pack()
details()
prepare()
PizzaStore
CreatePizza()
orderPizza()
Has a
Has a
Extends
Extends
NagpurPizzaStore
CreatePizza()
MumbaiPizzaStore
CreatePizza()
Extends
Extends
CheesePizza
CheesePizza()
prepare()
CornPizza
CornPizza()
prepare()
<<Interface>>
Interface
PizzaIngredientsFactory
CreateDough()
CreateSauce()
CreateCheese()
Has a
Sauce
display()
Extends
Extends
ChillySauce
display()
TomatoSauce
display()
implements
implements
NagpurPizzaIngredientsFactory
CreateDough()
CreateSauce()
CreateCheese()
MumbaiPizzaIngredientsFactory
CreateDough()
CreateSauce()
CreateCheese()
Has a
Dough
display()
Extends
Extends
ThickDough
display()
ThinDough
display()
Has a
Cheese
display()
Extends
Extends
CreamCheese
display()
MozzarellaCheese
display()

PAGE No.: 37
DATE :

Practical - 4

Aim: Write a program to implement Adapter design pattern.

Lab

Aim: Write a program for a duck adapter. The ducks have the properties of quaking and flying whereas turkeys can gobble & fly. Use adapter design pattern to adapt the behaviour of turkey to duck behaviour in order to use turkey objects in place of ducks.

Objectives: To converts the interface of a class into another interface.

- To make classes having incompatible interfaces to work together.

Theory:- An adapter design pattern acts as a connector between two incompatible interfaces that otherwise cannot be connected directly. An adapter wraps an existing class with a new interface so that it becomes compatible with the client's interface. It basically converts the interface of a class into another interface clients expect.

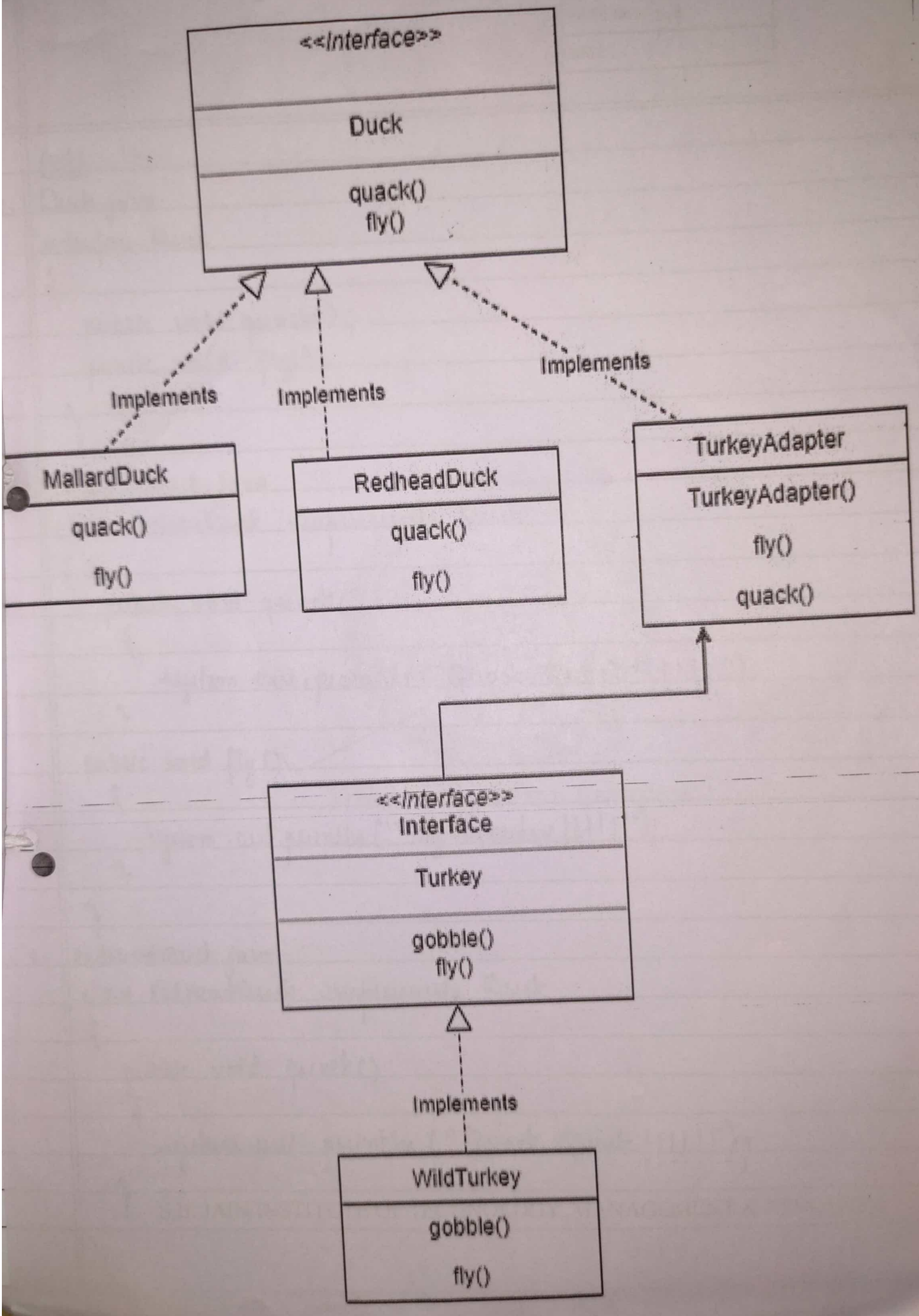
Here, to implement adapter design pattern we have considered an example of duck & turkey. First, we have created an interface of Duck with quack() & fly() operations. Each type of Duck is a subclass of Duck example MallardDuck & RedHeadDuck. These subclasses implement their own fly()

fly() & quack() operation.

Secondly, we have created Turkey interface which has similar methods as Duck. However, it has the gobble method instead of quack. They both have the fly method, but the behaviours may differ. Wild Turkey is the subclass of Turkey interface which defines gobble() & fly() method in it.

Now to use the Turkey instance as Duck, we have created an adapter which is a "TurkeyAdapter". This adapter will behave as Duck & will use a Turkey instance behind the scenes. Here, TurkeyAdapter implements the Duck interface, because we want to treat the Turkey object as Duck.

TurkeyAdapter also gets a reference to a backing Turkey object. This way, the adapter can forward calls to this object. For delegation purposes, TurkeyAdapter calls the fly method of Turkey five times. For the quack method, it just delegates to gobble. In this way we have converted Turkey interface into Duck interface & Turkey behaves like Duck now.


<<Interface>>
Duck
quack()
fly()
Implements
Implements
Implements
MallardDuck
quack()
fly()
RedheadDuck
quack()
fly()
TurkeyAdapter
TurkeyAdapter()
fly()
quack()
<<Interface>>
Interface
Turkey
gobble()
fly()
Implements
WildTurkey
gobble()
fly()

PAGE No.: 43

DATE :

Practical - 5

Aim: Write a program to implement decorator design pattern

Lab

Desc: Write a program for a coffee shop offering a variety of combinations of coffee with condiments. Cost of a cup depends on the combination which is offered. Use decorator design pattern to make this beverage.

Objective: To provide additional responsibilities to an object dynamically.

- To extend the system can without changing the existing code.

Theory:- Decorator pattern allows a user to add new functionality to an existing object without altering its structure. It attaches additional responsibilities to an object dynamically. So there is no change to the original class. The decorator design pattern is a structural pattern, which provides a wrapper to the existing class.

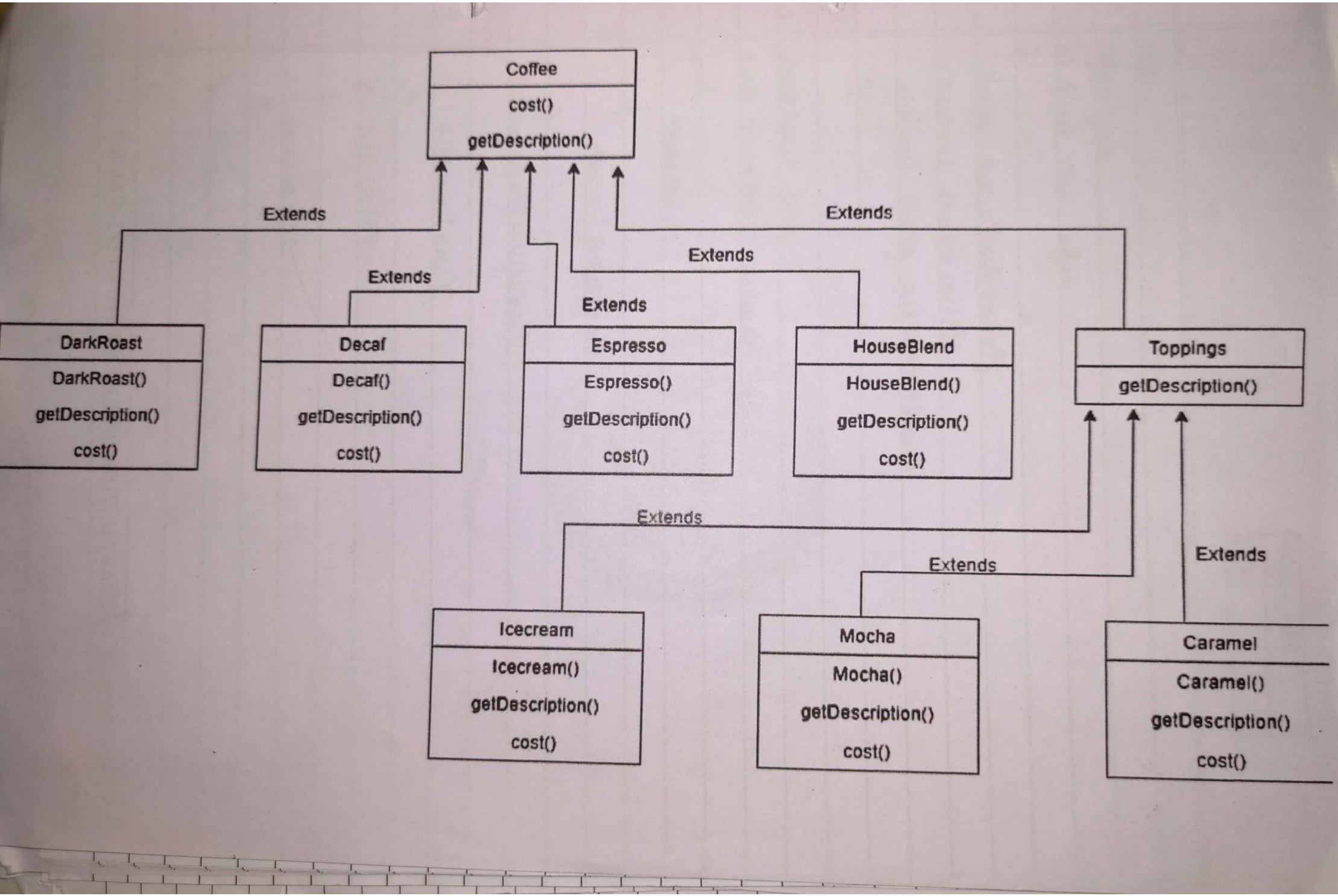
We have used the coffee shop abstraction to demonstrate the decorator pattern. We have created a coffee class with cost() & getDescription() methods & desc of coffee as "unknown". Then, Espresso, HouseBlend, Decaf & DarkRoast extends the coffee abstract class. They also implement the cost method & update the description. In getDescription method, we return the description of coffee & cost method returns cost of the coffee.

PAGE No.: 44

DATE :

After covering the actual coffees, we have created a Topping class which extends coffee. Toppings is the base class for all the additional toppings that we are going to add with the combination of coffee. Caramel, Icecream & mocha are the subclasses of Toppings in which we redefined the getDescription & cost method. Each of subclass of Toppings class contains a coffee instance. Additionally, each Toppings calls the wrapped coffee before / after completing its operation

Then we have created a main class "CD" for the actual implementation. where we have created object of coffee & Toppings. We have decorated Espresso instance with a caramel, Houseblend instance with a mocha & Decaf instance with an icecream.


Coffee
cost()
getDescription()
Extends
Extends
Extends
Extends
Extends
DarkRoast
DarkRoast()
getDescription()
cost()
Decaf
Decaf()
getDescription()
cost()
Espresso
Espresso()
getDescription()
cost()
HouseBlend
HouseBlend()
getDescription()
cost()
Toppings
getDescription()
Extends
Extends
Extends
Icecream
Icecream()
getDescription()
cost()
Mocha
Mocha()
getDescription()
cost()
Caramel
Caramel()
getDescription()
cost()

PAGE No.: 52
DATE :

Practical - 6.

Aim: Write a program to demonstrate the use of Facade and singleton pattern.

Prelab

Aim: Write a program for installing a home theatre consisting of several components like DVD player, CD player, Projector screen, theatre lights, popcorn popper, amplifier, Tuner, etc. The home theatre can be used to watch movies, listen to CD and listen to radio. Make use of Facade class in order to make theatre subsystem easy to use for a viewer. Since, there is a single facade declare Facade class as singleton.

Objectives
- To provide a unified interface to a set of interfaces in a subsystem.
- To defines a higher level interface that makes the subsystem easier to use.

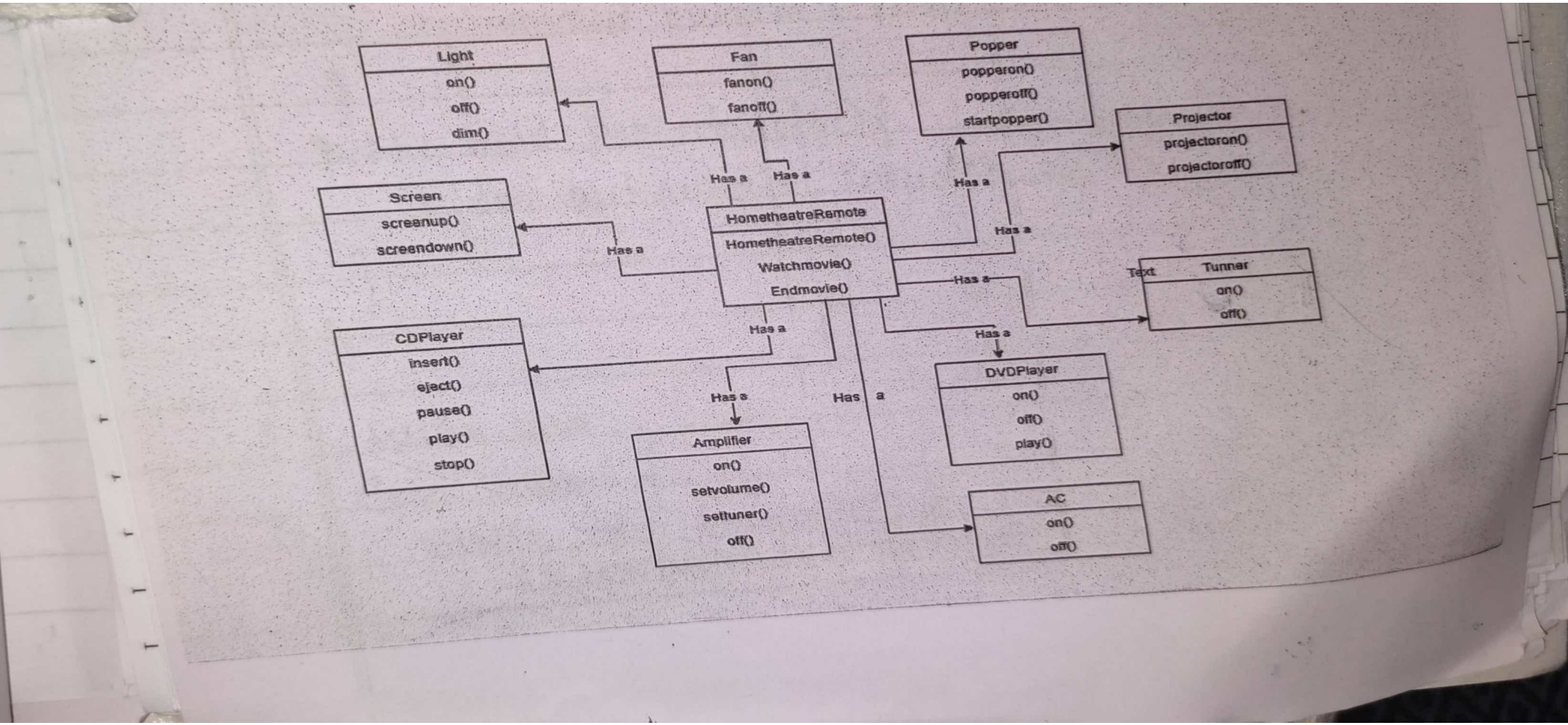
Theory: Facade is a structural design pattern that provides a simplified interface. The facade pattern says that "first provide a unified & simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client". In other words, Facade pattern describe a higher-level interface that makes the sub-system easier to use.

To implement facade pattern, we have taken an example of HomeTheater. Here, facade class treats the home theater components as a subsystem, and calls the subsystem to implement its watchmovie & endmovie method.

We have created different components of HomeTheater, we have created classes light, screen, projector, Fan, Popcornpopper, CDPlayer, DVDPlayer & Tuner. In these classes, we have defined different method like on(), off(), play(), pause(), etc. accordingly. In amplifier class we have created reference of CDPlayer, DVDplayer & Tuner & defined operations SetVolume(int level) & SetTuner(Tuner t).

Here, HomeTheater is a Facade class. As we are using only single facade, we have declared facade class as singleton. Then, created objects of all the components of Home Theater & defined Methods watchmovie() & endmovie().

In the ~~test~~ class, we have created reference of HomeTheater Him object using singleton patter and called the watchmovie & endmovie methods by using HomeTheater's object. In this way, we have made an interface for Home Theater as simple as possible.


Light
on()
off()
dim()
Fan
fanon()
fanoff()
Popper
popperon()
popperoff()
startpopper()
Projector
projectoron()
projectoroff()
Screen
screenup()
screendown()
HometheatreRemote
HometheatreRemote()
Watchmovie()
Endmovie()
Has a
Text Tunner
on()
off()
CDPlayer
insert()
eject()
pause()
play()
stop()
Amplifier
on()
setvolume()
settuner()
off()
DVDPlayer
on()
off()
play()
AC
on()
off()

PAGE No.: 64
DATE :

Practical No 7.

Aim: Write a program to demonstrate chain of responsibility design pattern.

Mulab

Aim: Write a program for ATM Dispense Handler: An ATM is an state machine which exhibits a different behaviour to a customer depending on its current state. A user enters amount in denomination of 100, 200, 500 and 2000 Use chain of responsibility design pattern to dispense appropriate count of bills for Rs 100, 200, 500 & 2000

Objective: • To avoid coupling ~~that~~ the sender of a request to it receiver

- To provide an object with more than one chance to handle the request
- To chain the receiving objects and pass the request along the chain until an object handles it.

Theory:- Chain of Responsibility is a behavioral design pattern that lets you pass requests along a chain of handlers Upon receiving a request, each handler decides whether to process the request or to pass it to the next handler in the chain.

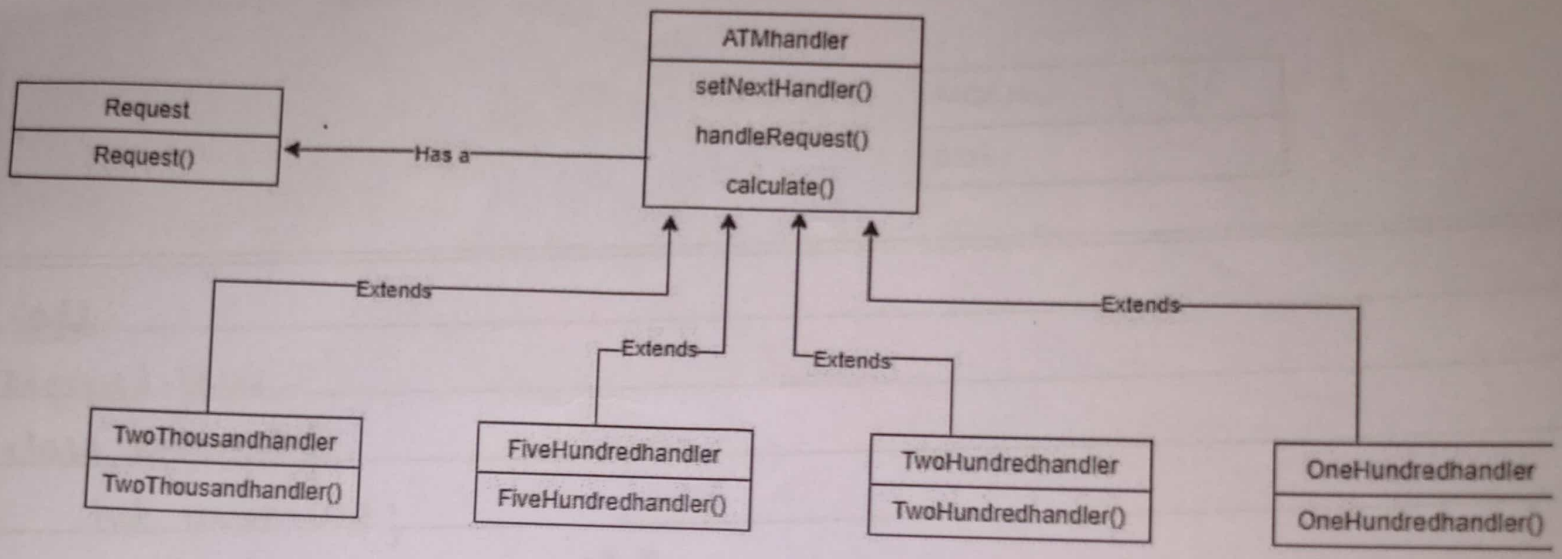
To implement chain of responsibility design pattern we have taken an example of ATM handler. An ATM uses the chain of responsibility design pattern in money giving

PAGE No.: 6.5
DATE :

in money giving process. First, we have created a "Request" class for request of amount from client. Then we have created "ATM Handler" class to handle the client request and dispatches the request to a chain of handlers. It has reference to the only first handler i.e 100 in the chain & does not know anything about the rest of the handlers.

Then we have created four concrete handlers i.e Two hundred Rupees Handler, Two Thousand Rupee Handler, Five Hundred Rupees Handler & Hundred Rupees Handler which extends from ATM Handler. These are actual handlers of the request chained in form sequencial order in this, concrete handlers we have assigned next as 2000, 500, 200 & 100.

For demonstration purpose, we have created "Demo" class. Declared static mainchain() method of ATM Handler type in which we have created objects of concrete handler classes & set the next handler using setNextHandler() method. Finally, in the main function client requested for the amount to be with draw & got the number of 2000, 500, 200 & 100 rupee note as per the requested amount.


ATMhandler
setNextHandler()
handleRequest()
calculate()
Request
Request()
Has a
Extends
Extends
Extends
Extends
TwoThousandhandler
TwoThousandhandler()
FiveHundredhandler
FiveHundredhandler()
TwoHundredhandler
TwoHundredhandler()
OneHundredhandler
OneHundredhandler()

PAGE No.: 70
DATE :

Practical - 8

Aim: Write a program to demonstrate Mediator design pattern

Solab

Aim: Write a program for chat room where multiple users can send message to chat room and it is the responsibility of chat room to show the message to all user. Use media for design pattern to share message between users using chat room.

Objectives:-
- To identify the importance of loose coupling between objects
- To define an object that encapsulate how a set of objects interact.

Theory: Mediator design pattern is one of the important and widely used behavioural design pattern. Mediator enables decoupling of objects by introducing a layer in between so that the interaction between objects happen via the layer if the objects interacts with each other that ~~matters~~ directly, the system components are highly-coupled with each other that makes higher maintainability cost and not need to extend. Mediator ~~Matl~~ pattern focuses on providing a mediator between objects for communication and help in implementing loose coupling between objects.

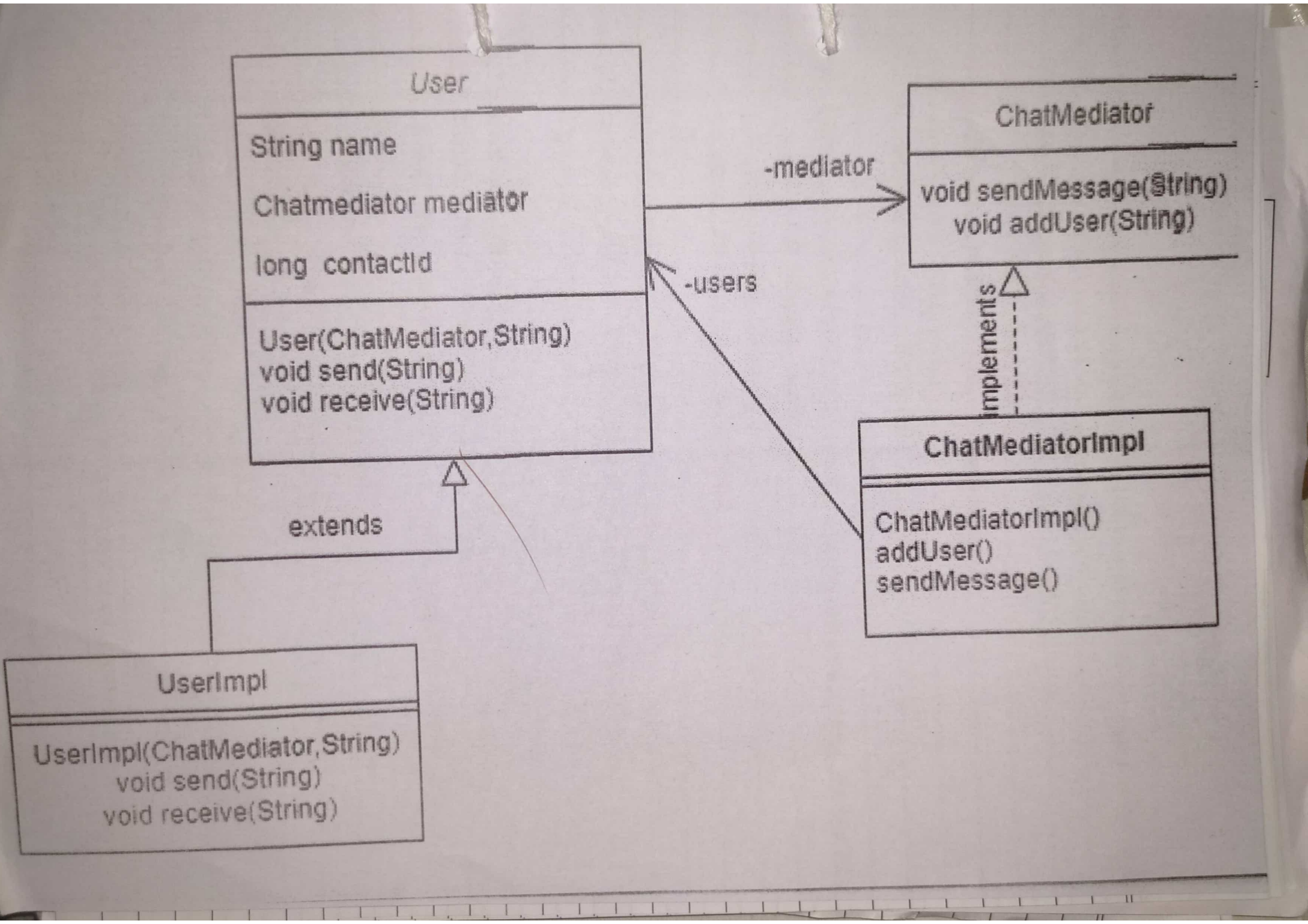
We have demonstrated Mediator pattern by example of a chat room where multiple users can send message to ~~that~~ chat room & it is the responsibility of chat room to show

message to all users. so, we have created User-User object with the chatroom method to share their message. User Impl extends from user & defined methods send() & received message.

Then created one "ChatMediator" interface from which we can send message using sendMessage() & add users using addUser() methods. ChatMediator Impl extends from ChatMediator & uses Array list to store & add the users. WhatsApp is the final class & in main() method we used the user object to show communication between them.

We have created an object to ~~show communication~~ of five different users along with their name and contact number and added them through the mediator. Then, user1 sent the message of "Good morning everyone" which is received by other remaining users. similarly, user3 sent Message of "Good Morning". Here the sender is not receiving message of himself / herself.

The Mediator design pattern is useful when the number of objects grows so large that it becomes difficult to maintain the references to the objects. The Mediator is essentially an object that encapsulates how one or more objects interact with each other.


User
String name
Chatmediator mediator
long contactId
User(ChatMediator,String)
void send(String)
void receive(String)
-mediator
ChatMediator
void sendMessage(String)
void addUser(String)
-users
implements
ChatMediatorImpl
ChatMediatorImpl()
addUser()
sendMessage()
extends
UserImpl
UserImpl(ChatMediator,String)
void send(String)
void receive(String)

Code

1. User.java

```
public abstract class User {
    protected ChatMediator mediator;
    protected String name;
    protected long contactId;

public User (ChatMediator med, String name, long contactId) {
    this.mediator = med;
    this.name = name;
    this.contactId = contactId;
}
public abstract void send (String msg);
public abstract void receive (String msg);
}
```

2. UserImpl.java

```
public class UserImpl extends User {
public UserImpl (ChatMediator med, String name, long contact)
{
    super (med, name, contact);
}
@Override
public void send (String msg) {
    System.out.println (this.name + " (" + this.contactId + "):
        Sending Message: \t\t" + msg);
System.out.println();
```

PAGE No.: 76
DATE :

Practical - 9

Aim: Write a program to demonstrate template method pattern.

Inlab

Inlab Aim: Write a program for starbuzz tea and coffee recipe. Use template method design pattern to define a generalized recipe as both are having the same steps and only two steps are having the different implementation.

Objectives:- • To define the skeleton of an algorithm in an operation

• To let the subclasses redefine certain steps an algorithm without changing the structure of algorithm.

Theory:-

Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes. The overall structure and sequence off the algorithm are preserved by the parent class

Template means preset format like HTML template which has fixed preset format.

Here, we have considered an example of starbuzz tea and coffee recipe, we created a class Beverage, in which we defined a skelton for the algorithm means

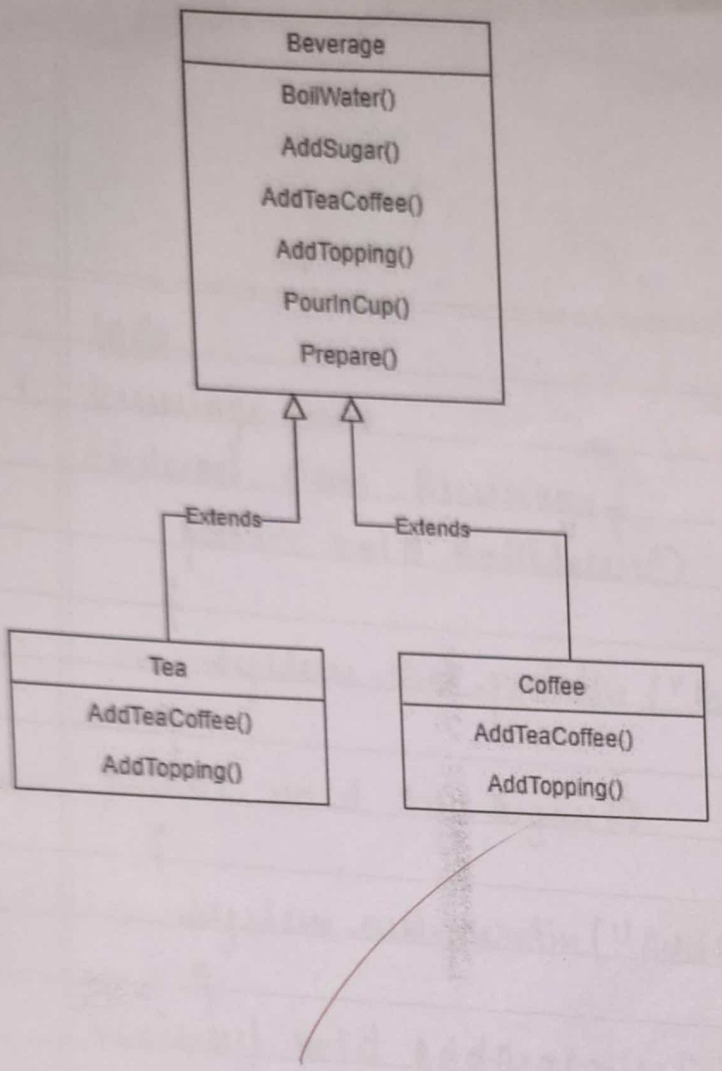
PAGE No.: 27
DATE :

a steps in a sequence for preparing a Beverage, Then created Tea class which extends from abstract class Beverage & defined abstract methods here.

Similarly, we have created coffee class which extends from abstract class Beverage with a hook, we can override a method or not, its our choice, if we don't then abstract class provides a default implementation.

Now let's create, coffee or tea, here customer needs to decide whether he/she wants condiment by an input. For this purpose, we have created test class in which we have created objects of Tea & Coffee & invoked method prepareBeverage as ~~for~~ ~~the~~ per need.

The template method is used in frameworks where each implements the invariants parts of domains architecture, leaving "placeholders" for customization workflow structure is implemented once in the abstract class algorithm and necessary variations are implemented in the subclasses.


Beverage
BoilWater()
AddSugar()
AddTeaCoffee()
AddTopping()
PourInCup()
Prepare()
Extends
Extends
Tea
AddTeaCoffee()
AddTopping()
Coffee
AddTeaCoffee()
AddTopping()

PAGE No.: 82
DATE :

Practical - 10

Aim: Write a program to demonstrate state design pattern

In Lab

Aim: Write a program for gumball machine in which there are total four states i.e Has quarter, No Quarter, Gumball sold, Out of gumball machine, changes its state use state pattern to show how gumball machine changes its state based on actions i.e insert quarter, turn crank, eject Quarter, dispense gumball.

Objectives: To enable the object to change its class.
- To allow an object to alter its behaviour when its internal state changes.

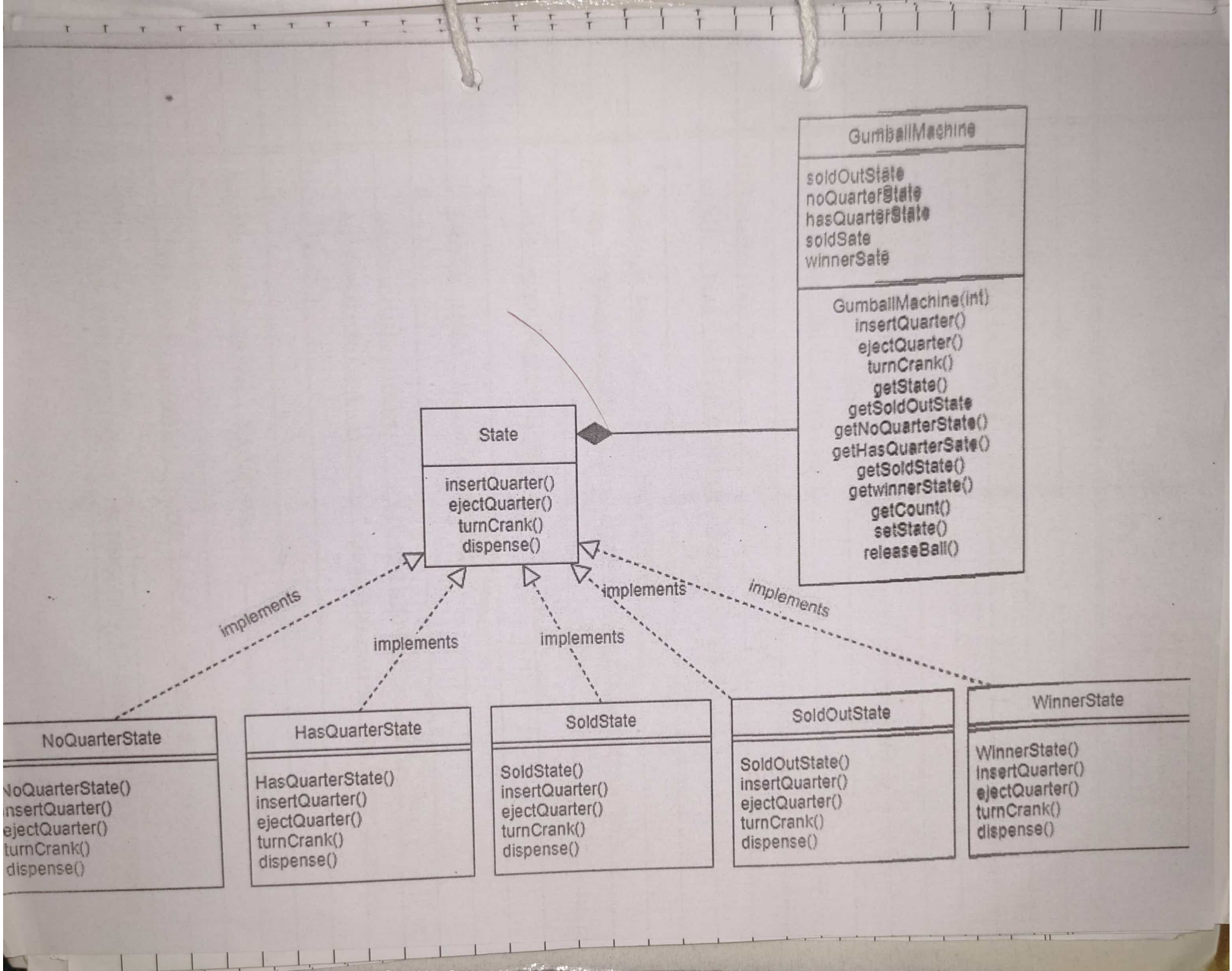
Theory:- State is a behavioral design pattern that allows an object to change the behavior when its internal state changes. The pattern extracts state-related behaviors into separate state classes and forces the original object to delegate the work to an instance of these classes, instead of acting on its own. The state information is determined at and can be changed runtime, which allows behaviours to be modified dynamically, making it as if the object has changed classes.

To implement state design pattern, we have taken an example of GumballMachine. It has different state that is represented by individual classes. So, we have created

four different classes NoQuarterState, HasQuarterState, SoldState, SoldOutState and one more mimmustate. In each of the classes, we have defined methods like insertQuarter(), ejectQuarter(), turnCrank() & dispense() accordingly each of these classes implements state interface that contains all these above methods.

Then, created "GumballMachine" class instantiate all concrete states & provide all possible action methods to the user based on states. Each state has a reference to the GumballMachine to transition it to a different state. When the user inserts a coin, the insertQuarter() Method uses the getState() & setState() of the GumballMachine class.

The final state is implemented in "sold out state" Then, to implement state design pattern we have created a GumballmachineTestDrive in which we invoked functions like insertQuarter() & turnCrank() & got the output according to the state of GumballMachine.


GumballMachine
soldOutState
noQuarterState
hasQuarterState
soldSate
winnerSate
GumballMachine(int)
insertQuarter()
ejectQuarter()
turnCrank()
getState()
getSoldOutState
getNoQuarterState()
getHasQuarterSate()
getSoldState()
getwinnerState()
getCount()
setState()
releaseBall()
State
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
implements
implements
implements
implements
implements
NoQuarterState
NoQuarterState()
nsertQuarter()
ejectQuarter()
turnCrank()
dispense()
HasQuarterState
HasQuarterState()
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
SoldState
SoldState()
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
SoldOutState
SoldOutState()
insertQuarter()
ejectQuarter()
turnCrank()
dispense()
WinnerState
WinnerState()
InsertQuarter()
ejectQuarter()
turnCrank()
dispense()

PAGE No.: 95
DATE :

Practical No 11

Aim: To solve object oriented design problem using design pattern

In lab

Aim: Patterns are often used together and combined with the same design solution. A compound pattern combines two or more patterns into solution that solves a recurring or general problem. Considering this fact identify the design patterns that we can use in design of a effective, extendible and flexible Duck simulation game.

Objectives:- • To solve the object oriented design problem by combining two or more patterns.

Theory:- To implement this practical, we are combining two design pattern i.e Strategy pattern & Adapter pattern. Strategy is a behavioural design pattern that turns a set of behaviours into object. Adapter is a structural design pattern, which allows incompatible objects to collaborate. It converts one interface into other in

Here we have taken an example of Duck. In strategy pattern, we have divided different techniques & in adapter pattern we have converted turkey & interface into Duck interface. In this practical, we combined both & designed a Duck simulation game.

First we have created an interface IDuck in which we have declared two methods fly() and quack() then two concrete classes MallardDuck & RedHeadDuck that implements

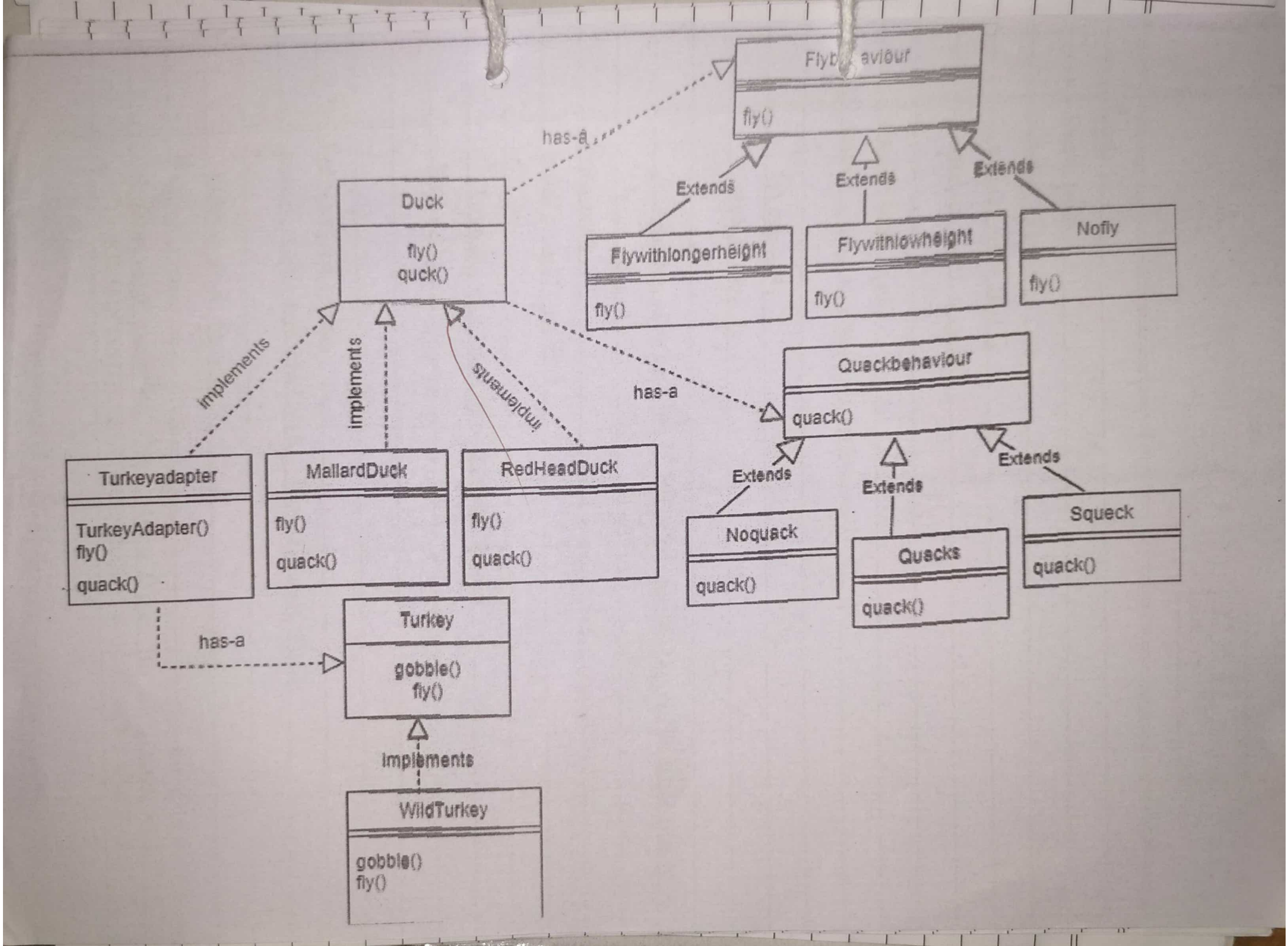
PAGE No.: 56
DATE :

Duck & defined fly() & quack() method in both the classes.

We have created a separate classes of Duck Behaviour technique. i.e FlyBehaviour class & QuackBehaviour class. FlyWithLongerHeight, FlyWithShortHeight & NoFly are the subclasses of flybehaviour in which we have redefined fly() method. Similarly, NOQuack, Quacks and Squeak are the subclasses of QuackBehaviour in which we have defined quack() method.

Also created another interface "Turkey" with gobble() & fly() operations. WildTurkey implements Turkey with gobble() & fly() methods defined in it. TurkeyAdapter acts as an adapter between Quack Duck & Turkey & invoke fly() & gobble() methods using Turkey reference.

Then created final class to execute the program in which we have created an object of WildTurkey & TurkeyAdapter(wt) by passing the reference of Turkey. Invoked the method testduck(d) by using Duck reference & created RedHeadDuck & MallardDuck object d1 & d2. passing d1, & d2 in testduck & called methods fly() & quack() in testduck(Duck d).


Flybehaviour
fly()
has-a
Extends
Extends
Extends
Duck
fly()
quck()
Flywithlongerheight
fly()
Flywithlowheight
fly()
Nofly
fly()
implements
implements
implements
has-a
Quackbehaviour
quack()
Turkeyadapter
TurkeyAdapter()
fly()
quack()
MallardDuck
fly()
quack()
RedHeadDuck
fly()
quack()
Extends
Extends
Extends
Noquack
quack()
Quacks
quack()
Squeck
quack()
has-a
Turkey
gobble()
fly()
Implements
WildTurkey
gobble()
fly()

PAGE No.: 103
DATE :

Practical No: 12

Aim: Write a program to implement MVC design pattern in LAB

Aim: WAP to implement MVC design pattern for timetable the information in order to make a timetable object acting as a model. Timetableview will be a view class which can print timetable details on console and timetable controller is the controller class responsible to store data in timetable model object and update view timetableview accordingly.

Objectives: To specifies that an application consists of a data model, presentation information and control information.

Theory:- The model view controller (MVC) design pattern specifies that an application consist of a data model, presentation information and control information. The pattern requires that each of these be separated into different objects

- The model contains only the pure application data it, contains no logic describing how to represent the data to a user
- The view presents the model's data to the user. The view knows how to access the model's data but it does not know what this data means
- The controller exists between the view & the model It listen to event triggered by the view and execute appropriate

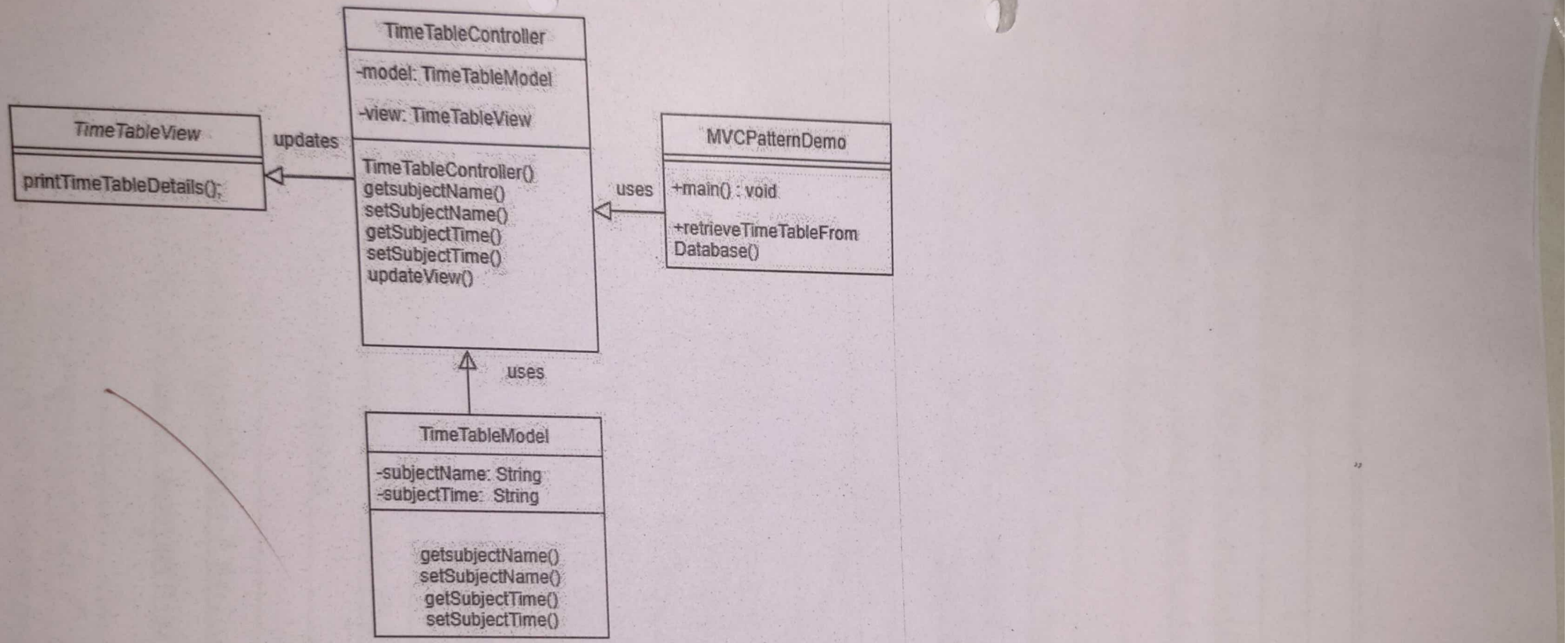
~~execution~~ creation to these events.

To demonstrate MVC design patterns, we have taken an example of timetable, where we have to print subjectname & subject Time. For this, we have created three classes i.e Time table model as a model class, timetableview as a view class & Timetable controller as a controller class. Add one class i.e MVC pattern demo to execute the program.

TimeTableView class is only to print details on the console. We haven't put any logic in the view class printTimeTableDetails() method is defined to print Subject Name & Subject Time.

TimeTableModel class is for setter & getter methods. This class is only for getting & setting the to represent the data.

Then all the logic is present in the controller class i.e Timetable controller. It controls all the events triggered by the view class. The methods are defined here to set SubjectName according to SubjectTime. Based on the conditions, the subjectname & subject time will be set & by creating an object of model & view class and having different methods by passing appropriate values the view get TimeTabledetails on the console. This is how we have demonstrated MVC design pattern.


TimeTableController
-model: TimeTableModel
-view: TimeTableView
TimeTableController()
getsubjectName()
setSubjectName()
getSubjectTime()
setSubjectTime()
updateView()
TimeTableView
printTimeTableDetails();
updates
MVCPatternDemo
+main() : void
+retrieveTimeTableFrom Database()
uses
uses
TimeTableModel
-subjectName: String
-subjectTime: String
getsubjectName()
setSubjectName()
getSubjectTime()
setSubjectTime()