Project Final Report

CS586 – Software System Architecture

April 25 2018

NIKITA JADHAV

A20401223

1. MDA-EFSM model for the GasPump components

i. MDA-EFSM Events:

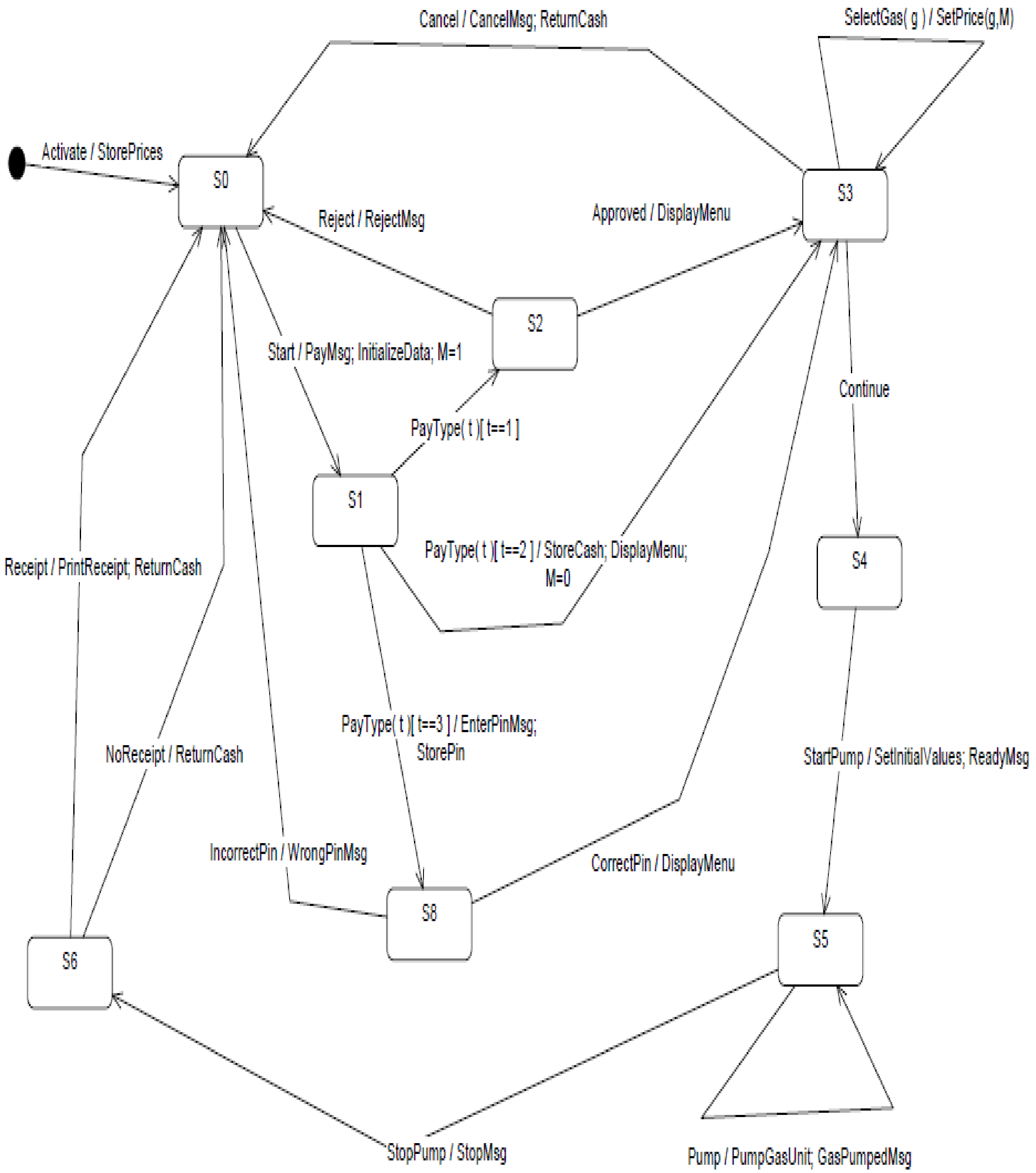
Activate()
Start()
PayType(int t) //credit: t=1; cash: t=2; debit: t=3
Reject()
Cancel()
Approved()
StartPump()
Pump()
StopPump()
SelectGas(int g) // Regular: g=1; Super: g=2; Premium: g=3; Diesel: g=4
Receipt()
NoReceipt()
CorrectPin()
IncorrectPin()
Continue()

ii. MDA-EFSM Actions:

StorePrices // stores price(s) for the gas from the temporary data store
PayMsg // displays a type of payment method
StoreCash // stores cash from the temporary data store
DisplayMenu // display a menu with a list of selections
RejectMsg // displays credit card not approved message

SetPrice(int g, int M) // set the price for the gas identified by g
identifier as in SelectGas(int g); if M=1, the price may be increased
ReadyMsg // displays the ready for pumping message
SetInitialValues // set G (or L) and total to 0;
PumpGasUnit   // disposes unit of gas and counts # of units
disposed GasPumpedMsg   // displays the amount of disposed gas
StopMsg   // stop pump message and receipt? msg (optionally)
PrintReceipt   // print a receipt
CancelMsg   // displays a cancellation message
ReturnCash   // returns the remaining cash
WrongPinMsg   // displays incorrect pin message
StorePin   // stores the pin from the temporary data store
EnterPinMsg   // displays a message to enter pin
InitializeData   // set the value of price and cash to 0

iii. A state diagram of the MDA-EFSM


Cancel / CancelMsg; ReturnCash
SelectGas(g) / SetPrice(g,M)
Activate / StorePrices
S0
Reject / RejectMsg
Approved / DisplayMenu
S3
S2
Start / PayMsg; InitializeData; M=1
Continue
PayType(t)[t==1]
S1
PayType(t)[t==2] / StoreCash; DisplayMenu;
M=0
S4
Receipt / PrintReceipt; ReturnCash
PayType(t)[t==3] / EnterPinMsg;
StorePin
NoReceipt / ReturnCash
StartPump / SetInitialValues; ReadyMsg
IncorrectPin / WrongPinMsg
CorrectPin / DisplayMenu
S8
S5
S6
StopPump / StopMsg
Pump / PumpGasUnit; GasPumpedMsg

MDA-EFSM for Gas Pumps

iv. Pseudocode

Operations of the Input Processor (GasPump-1)

```
Activate(float a, float b) { if ((a>0)&&(b>0)) {
d->temp_a=a; d->temp_b=b; m->Activate()
}
}

Start() {
m->Start();
}

PayCredit() {
m->PayType(1);
}

Reject() {
m->Reject();

Approved() {
m->Approved();
}

Diesel() {
m->SelectGas(4)
}

Regular() {
m->SelectGas(1)
}

StartPump() {
if (d->price>0) {
m->Continue(); m->StartPump();
}
}
```

```
PumpGallon() {
m->Pump();

}

PayDebit(string p) {
d->temp_p=p; m->PayType(3);


StopPump() {
m->StopPump(); m->Receipt();
}

}

Pin(string x) {
if (d->pin==x) m->CorrectPin() else m->InCorrectPin();


FullTank() {
m->StopPump(); m->Receipt();
}

}

Cancel() {
m->Cancel();

}
```

Operations of the Input Processor (GasPump-2)

```
Activate(int a, int b, int c) {
if ((a>0)&&(b>0)&&(c>0)) { d->temp_a=a;
d->temp_b=b; d->temp_c=c m->Activate()
}

Super() {
m->SelectGas(2); m->Continue();
}
```

```
Premium() {
m->SelectGas(3); m->Continue();
}
```

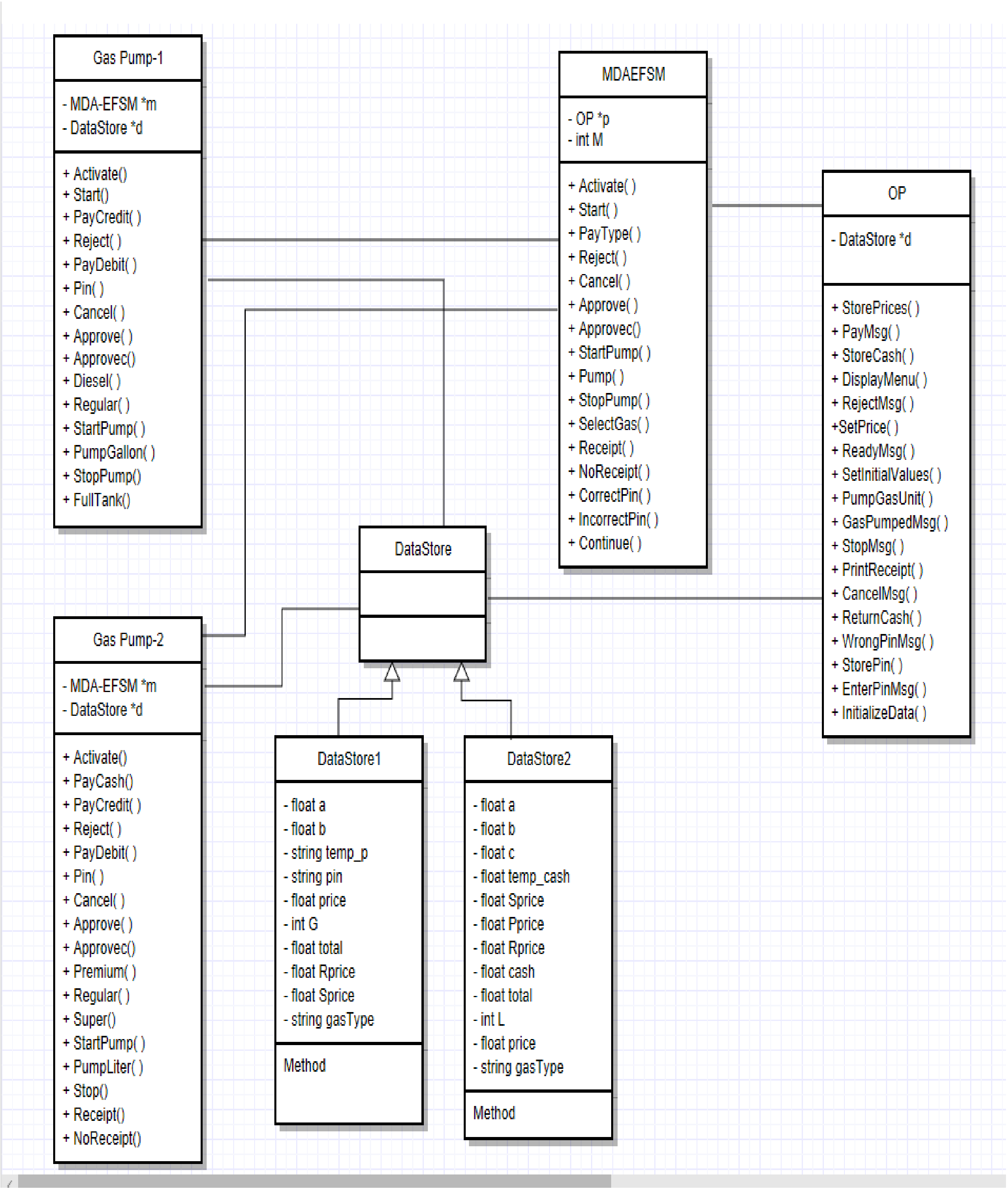
Notice:
cash: contains the value of cash deposited price: contains the price of the selected gas L: contains the number of liters already pumped

cash , L, price are in the data store
m: is a pointer to the MDA-EFSM object
d: is a pointer to the Data Store object

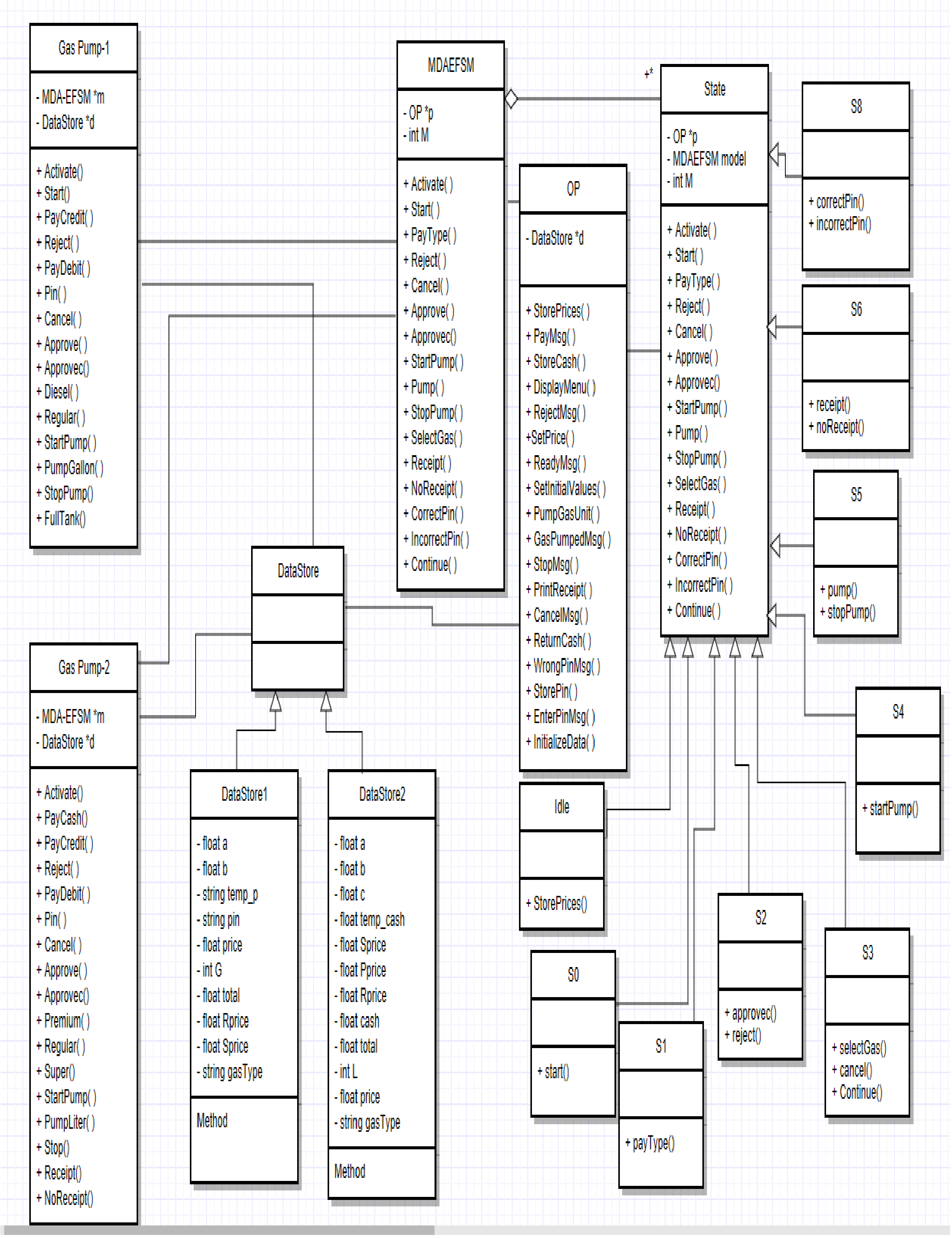
```
}

PayCash(float c) {
if (c>0) {
d->temp_cash=c; m->start();
m->PayType(2)
}
}

PayCredit() {
m->start();
m->PayType(1);
}

Reject() {
m->Reject();
}

Approved() {
m-> Approved();
}
Cancel() {
m->Cancel();
}


Regular() {
m->SelectGas(1); m->Continue();
```

```
}

StartPump() {
m->StartPump();
}

PumpLiter() {
if (d->cash>0)&&(d->cash < d->price*(d->L+1))
m->StopPump(); else m->Pump()
}

Stop() {
m->StopPump();
}

Receipt() {
m->Receipt();
}

NoReceipt() {
m->NoReceipt();

}
```

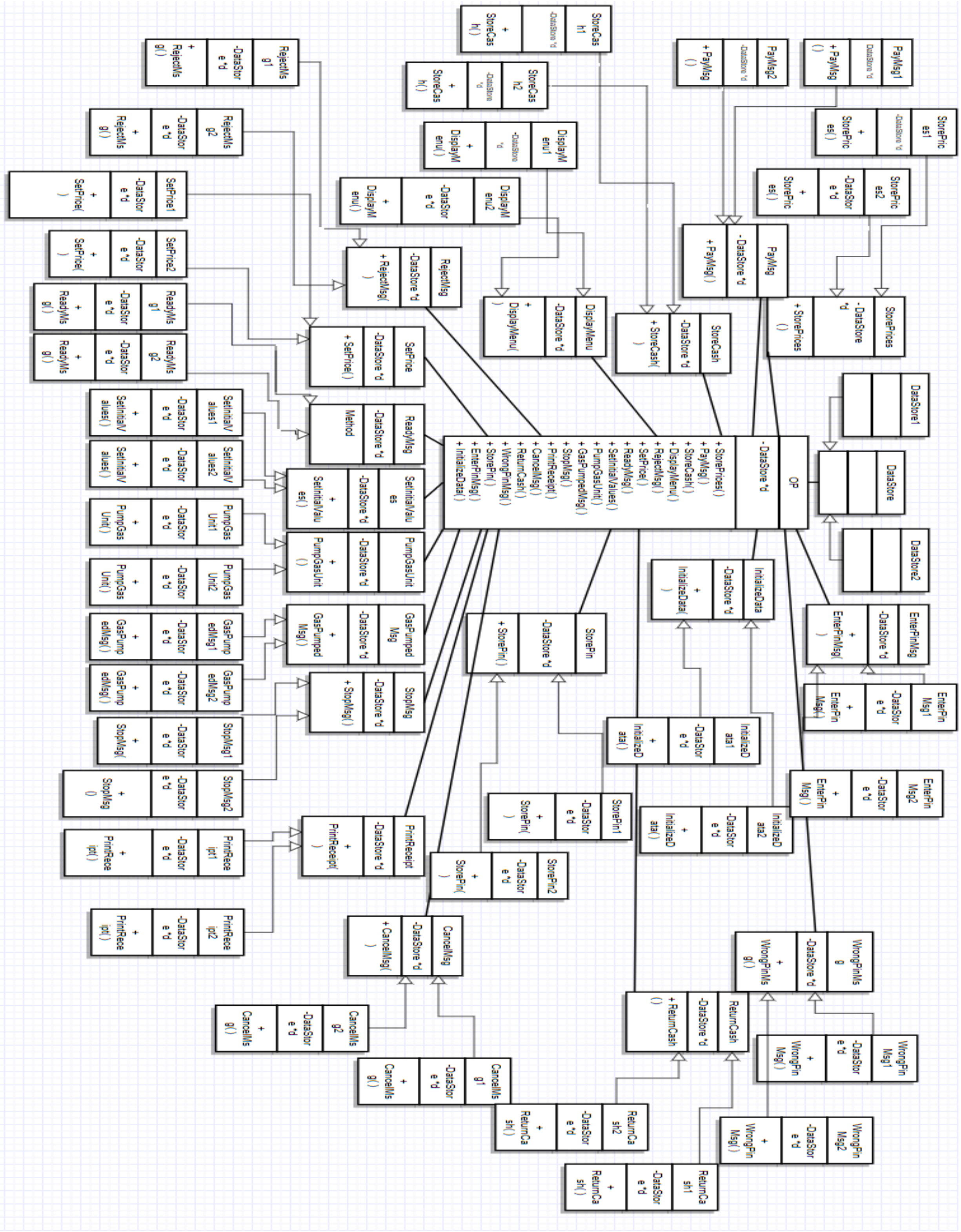
2. Class diagram(s) of the MDA of the GasPump components. Overall Class Diagram



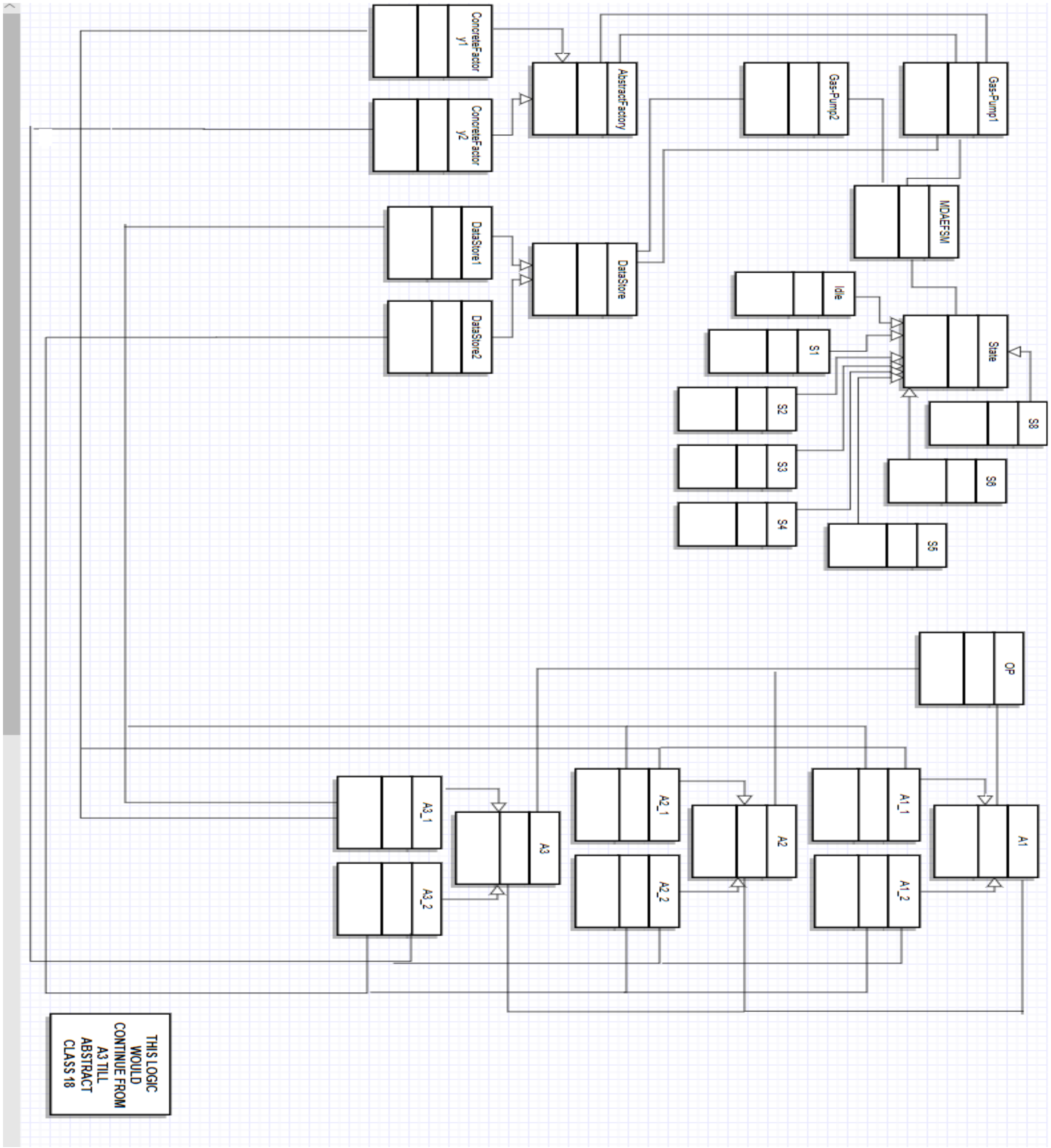
i. State Pattern Diagram



ii. Strategy Pattern



iii. Abstract Factory Pattern



3. Purpose, Attributes, and Functions of all the classes

Class GasPump

Class GasPump1

Purpose and Attributes	Represents GasPump1 implementation model is object for MDAEFSM data is object for DataStore

Activate (float a, float b)	**if** a > 0 && b > 0
	   d.a = a;
	   d.b = b;
	   model.activate();
	else
	   print (“Price should be > 0”)

Start ()	model.Start ();
PayDebit()	Model.payType(3);
PayCredit ()	model.payType (1);

Pin()	d.temp_p=x;
	d.pin=d.temp_p;

Reject ()	model.reject ();
Cancel ()	model.cancel ();

Approve (String p)	if(p.equals(d.pin))
	       {
	          model.correctPin();
	       }
	       else
	       {
	          model.incorrectPin();
	       }

Approvec()	model.approvec();
Diesel ()	model.selectGas (4,1);
Regular ()	model.selectGas (1,1);

StartPump ()	model.Continue();
	        if(d.price>0)
	        {
	        model.startPump();
	        }

PumpGallon ()	model.pump ();

StopPump ()	model.stopPump ();
	model.receipt ();

FullTank()	Model.receipt();

Class GasPump2

Purpose and Attributes	Represents GasPump2 implementation model is object for MDAEFSM data is object for DataStore

Activate (int a, int b, int c)	**if** a > 0 && b > 0 && c > 0
	d.a = a;
	d.b = b;
	d.c = c;
	model.activate();
	else
	print (“Price should be > 0”)

Start ()	model.Start ();

PayCash ()	**if** cash > 0
	d.temp_cash = cash;
	model.payType(2);
	d.M=0;
	else
	print (“Cash should be > 0”)

PayCredit()	**model.payType(1);**
	d.M=1;

Reject()	**Model.reject();**
Cancel ()	model.Cancel ();

Premium ()	if(d.M==1)
	{
	model.selectGas(3,1);
	}
	else
	model.selectGas(3, 0);

Regular ()	if(d.M==1)
	{
	model.selectGas(1,1);
	}
	else
	model.selectGas(1, 0);

Super ()	if(d.M==1)
	{
	model.selectGas(2,1);
	}
	else
	model.selectGas(2, 0);

StartPump ()	model.Continue();
	if(d.M==1)
	{

	model.startPump();
	d.price= (float)(1.1 * d.price);
	d.L=0;
	
	}
	else if(d.M==0)
	{
	model.startPump();
	d.L=0;
	}

PumpLiter ()	**if ((d.M==0) && (d.cash < d.price * (d.L + 1)))**
	{
	System.out.println("Limited Balance");
	model.stopPump();
	}
	else if((d.M==0) && (d.cash>=d.price * (d.L + 1)))
	{
	model.pump();
	
	}
	else if((d.M==1))
	{
	model.pump();
	}

Approve()	**Model.approve()**
	d.M=1;

Approvec()	**Model.approvec()**
Stop ()	model.stopPump ();
Receipt ()	model.Receipt ();
NoReceipt ()	model.NoReceipt ();

State Pattern

Class MDAEFSM

Purpose and Attributes	This class implements the common functionality among two GasPumps. f is an object of AF (Abstract Factory). op is an object of OP.
MDAEFSM	It is the constructor. It sets the current state of the GasPump to Start
Activate()	Call State Function Activate()
Start()	Call State Function Start()
PayType(int g)	Call State Function PayType(int g)
Approve()	Call State Function Approve()

Approvec()	Call State Function Approvec()
Cancel()	Call State Function Cancel()
StartPump()	Call State Function StartPump()
Pump()	Call State Function Pump()
StopPump()	Call State Function StopPump()
SelectGas(int g, int M)	Call State Function SelectGas(int g, int M)
Receipt()	Call State Function Receipt()
NoReceipt()	Call State Function NoReceipt()
Reject()	Call State Function Reject()
Continue()	Call State Function Continue()
correctPin()	Call State Function correctPin()
incorrectPin()	Call State Function incorrectPin()

Class Idle

Purpose and Attributes	This is the concrete class for the Idle State. Idle() is the class constructor.
Activate()	Calls StorePrices() to store price of gases in temporary variables and set the state to S0

Class S0

Purpose and Attributes	This is the concrete class for the S0 State. S0() is the class constructor.
Start()	Calls PayMsg() Displays a pay message Calls InitializeData() M=1 Set the state to S1

Class S1

Purpose and Attributes	This is the concrete class for the S1 State. S1() is the class constructor.
PayType(int t)	If (PayType == 1 ie credit and state is S1) Then set state to S2 Set M=1 Else if (PayType == 2 ie cash and state is S1) {Then call StoreCash() to store the cash inserted value in Temp_cash. Call DisplayMenu() to display menu of operations of gaspump1 Set state to S3 Set M=0

	} Else if(PayType == 3 ie debit and state is S1) {Then call EnterPinMsg() & StorePin() Set state to S8 Set M=1 }

Class S2

Purpose and Attributes	This is the concrete class for the S2 State. S2() is the class constructor.
Approvec()	If state is in S2 then Sets state to S3 Calls DisplayMenu() to display menu of operations of gaspump1
Reject()	If state is in S2 then Sets state to S0 Calls RejectMsg() to display a reject message

Class S3

Purpose and Attributes	This is the concrete class for the S3 State. S3() is the class constructor.
SelectGas(int g)	If state is in S3 then Sets state to S3 Calls SetPrice(int g,int M) to store price of selected gas in price parameter from temp_cash
Cancel()	If state is in S3 then Sets state to S0 Calls CancelMsg() to display a cancel message Calls ReturnCash() to return the left cash
Continue()	If state is in S3 then Set state to S4

Class S4

Purpose and Attributes	This is the concrete class for the S4 State. S4() is the class constructor.
StartPump()	If state is in S4 then Sets state to S5 Calls SetInitialValues() to set value of G/L and total to 0 Calls ReadyMsg() to display that gaspump is ready to pump

Class S5

Purpose and Attributes	This is the concrete class for the S5 State.

	S5() is the class constructor.
Pump()	If state is in S5 then Calls PumpGasUnit() to increments value of G by 1 and calculates the value of total and GasPumpedMsg() to display the gallons of disposed gas
StopPump()	If state is in S5 then it sets state to S6 Calls StopMsg() to display a stop message Calls PrintReceipt() to print the receipt

Class S6

Purpose and Attributes	This is the concrete class for the S6 State. S6() is the class constructor.
Receipt()	If state is in S6 then it sets state to S0 Calls PrintReceipt() to print receipt Calls ReturnCash() to return cash
NoReceipt()	If state is in S6 then it sets state to S0 Calls ReturnCash() to return cash

Class S8

Purpose and Attributes	This is the concrete class for the S8 State. S8() is the class constructor.
correctPin()	If state is in S8 then it sets state to S3 Calls DisplayMenu()
incorrectPin()	If state is in S8 then it sets state to S0 Calls WrongPinMsg()

Strategy Pattern:

Class StoreData

Purpose and Attributes	This class represents the abstract factory class of action StoreData. data is an object of DataStore
StoreData()	Abstract method

Class StoreData1

Purpose and Attributes	This class represents the concrete class for StoreData action in GasPump1 and is used to store the values of temporary variables a and b
StoreData()	Stores values of temporary variables a and b

Class StoreData2

Purpose and Attributes	This class represents the concrete class for StoreData action in GasPump2 and is used to store the values of temporary variables a, b and c
StoreData()	Stores values of temporary variables a, b and c

Class PayMsg

Purpose and Attributes	This class represents the abstract class for PayMsg action
PayMsg()	Abstract method

Class PayMsg1

Purpose and Attributes	This class represents the concrete class for PayMsg action for GasPump1.
PayMsg()	Ask for the payment type from user for GasPump1

Class PayMsg2

Purpose and Attributes	This class represents the concrete class for PayMsg action for GasPUmp2.
PayMsg()	Ask for the payment type from the user for GasPump2

Class StoreCash

Purpose and Attributes	This class represents the abstract factory class of action StoreCash. data is an object of DataStore
StoreCash()	Abstract method

Class StoreCash1

Purpose and Attributes	This class is not supported by GasPump1 as it does not support payment by cash.
StoreCash()	No action

Class StoreCash2

Purpose and Attributes	This class represents the concrete class for action StoreCash GasPump2 version and is used to store cash in temp_cash in float format.

StoreCash()	Stores cash inserted into the GasPump2

Class DisplayMenu

Purpose and Attributes	This class represents the abstract class for DisplayMenu action
DisplayMenu()	Abstract method

Class DisplayMenu1

Purpose and Attributes	This class is used to represent the concrete class for DisplayMenu action for GasPump1
DisplayMenu()	Display Menu for GasPump1

Class DisplayMenu2

Purpose and Attributes	This class is used to represent the concrete class for DisplayMenu action for GasPump2
DisplayMenu()	Display Menu for GasPump2

Class RejectMsg

Purpose and Attributes	This class represents the abstract class for RejectMsg action
RejectMsg()	Abstract method

Class RejectMsg1

Purpose and Attributes	This class is used to represent the concrete class for RejectMsg action for GasPump1
RejectMsg()	Display credit card rejected message.

Class RejectMsg2

Purpose and Attributes	This class is not supported by GasPump2 as it does not support payment by credit card.
RejectMsg()	No action

Class SetPrice

Purpose and Attributes	This class represents the abstract factory class of action SetPrice. data is an object of DataStore

SetPrice()	Abstract method

Class SetPrice1

Purpose and Attributes	This class is used to represent the concrete class for SetPrice action for GasPump1 and is used to set the value of Rprice and Sprice as per the values entered by users for a and b respectively.
SetPrice()	Assign values to Rprice and Sprice

Class SetPrice2

Purpose and Attributes	This class is used to represent the concrete class for SetPrice action for GasPump2 and is used to set the value of Rprice, Sprice and Pprice as per the values entered by users for a, b and c respectively.
SetPrice()	Assign values to Rprice, Sprice and Pprice.

Class ReadyMsg

Purpose and Attributes	This class represents the abstract class for action ReadyMsg. data is an object for DataStore
ReadyMsg()	Abstract method

Class ReadyMsg1

Purpose and Attributes	This class represents the concrete class for ReadyMsg action for GasPump1 data is an object for DataStore
ReadyMsg()	Call DataStore1 to increment the value of G. Display ready to dispense message.

Class ReadyMsg2

Purpose and Attributes	This class represents the concrete class for ReadyMsg action for GasPump2 data is an object for DataStore
ReadyMsg()	Call DataStore2 to increment the value of L. Display ready to dispense message.

Class SetInitialValues

Purpose and Attributes	This class represents the abstract factory class of action SetInitialValues. data is an object of DataStore
SetInitialValues()	Abstract method

Class SetInitialValues1

Purpose and Attributes	This class represents the concrete class for SetInitialValues action of GasPump1. data is an object for DataStore1
SetInitialValues()	Sets the initial values of G and total to 0.

Class SetInitialValues2

Purpose and Attributes	This class represents the concrete class for SetInitialValues action of GasPump2. data is an object for DataStore2
SetInitialValues()	Sets the initial values of L and total to 0.

Class PumpGasUnit

Purpose and Attributes	This class represents the abstract factory class for PumpGasUnit action. data is an object of DataStore
PumpGasUnit()	Abstract method

Class PumpGasUnit1

Purpose and Attributes	This class represents the concrete class for PumpGasUnit for GasPump1. Fetches the values for G and total from DataStore1
PumpGasUnit()	Calculates and update values of G and total.

Class PumpGasUnit2

Purpose and Attributes	This class represents the concrete class for PumpGasUnit for GasPump2. Fetches the values for G and total from DataStore2
PumpGasUnit()	Calculates and update values of L and total.

Class GasPumpedMsg

Purpose and Attributes	This class represents the abstract class for GasPumpedMsg action. data is an object for DataStore
GasPumpedMsg()	Abstract method

Class GasPumpedMsg1

Purpose and Attributes	This class represents the concrete class for GasPumpedMsg action. It fetches the type of gas and total number of gallons from DataStore1 and displays the same.
GasPumpedMsg()	Displays type of gas and total gallons dispensed.

Class GasPumpedMsg2

Purpose and Attributes	This class represents the concrete class for GasPumpedMsg action. It fetches the type of gas and total number of liters from DataStore1 and displays the same.
GasPumpedMsg()	Displays type of gas and total liters dispensed.

Class StopMsg

Purpose and Attributes	This class represents the abstract factory class for StopMsg action.
StopMsg()	Abstract method

Class StopMsg1

Purpose and Attributes	This class represents the concrete class for action StopMsg for GasPump1.
StopMsg()	Display GasPump Stopping for GasPump1.

Class StopMsg2

Purpose and Attributes	This class represents the concrete class for action StopMsg for GasPump2.
StopMsg()	Display GasPump Stopping for GasPump2.

Class PrintReceipt

Purpose and Attributes	This class represents the abstract factory class of PrintReceipt action. data is an object of DataStore
PrintReceipt()	Abstract method

Class PrintReceipt1

Purpose and Attributes	This class represents the concrete class for PrintReceipt action for GasPump1. It generates and prints the receipt for GasPump1 by fetching the values of G, GasType and total gallons.
PrintReceipt()	Generates and displays receipt which includes no. of gallons of gasType and the total amount.

Class PrintReceipt2

Purpose and Attributes	This class represents the concrete class for PrintReceipt action for GasPump2. It generates and prints the receipt for GasPump2 by fetching the values of G, GasType and total liters and cash inserted by user.
PrintReceipt()	Generates and displays receipt which includes no. of liters of gasType and the total amount along with the cash amount inserted by user.

Class CancelMsg

Purpose and Attributes	This class represents the abstract factory class for CancelMsg action
CancelMsg()	Abstract method

Class CancelMsg1

Purpose and Attributes	This class represents the concrete class for CancelMsg action for GasPump1.
CancelMsg()	Display Transaction cancellation message for GasPump1.

Class CancelMsg2

Purpose and Attributes	This class represents the concrete class for CancelMsg action of GasPump2.
CancelMsg()	Display Transaction cancellation message for GasPump2.

Class ReturnCash

Purpose and Attributes	This class represents the abstract factory class for ReturnCash action. data is an object of DataStore
ReturnCash()	Abstract method

Class ReturnCash1

Purpose and Attributes	This class is not supported by GasPump1 as it does not support payment by cash.
ReturnCash()	No action.

Class ReturnCash2

Purpose and Attributes	This class represents the concrete class for ReturnCash action for GasPump2 in float format. It fetches the data for cash and total from DataStore2 and computes the difference of cash and total.
ReturnCash()	Displays returning cash message if the difference > 0 else it displays no cash to return message.

Abstract Factory Pattern

Class AbstractFactory:

Purpose and Attributes	Represents the Abstract class for factory that has different classes of GasPump
AbstractFactory	Create objects for MDAEFSM actions.

Class ConcreteFactory1:

Purpose and Attributes	Represents the concrete class for GasPump1's factory; used to handle creation of class objects specific to GasPump1
Actions	Create objects to every Strategy classes for GasPump1. Return objects of every Strategy class for GasPump1.

Class ConcreteFactory2:

Purpose and Attributes	Represents the concrete class for GasPump2's factory; used to handle creation of class objects specific to GasPump2
Actions	Create objects to every Strategy classes for GasPump2. Return objects of every Strategy class for GasPump2.

Output

Class Output:

Purpose and Attributes - Op is for output and it represents the Output processor of the MDA.

- data is object of Data Store and af is object of Abstract Factory.

Op ()	Constructor and pass Object of Abstract Factory
cancelMsg ()	Create factory object and call the CancelMsg strategy class function CancelMsg()
displayMenu ()	Create factory object and call the DisplayMenu strategy class function DisplayMenu()
gasPumpedMsg ()	Create factory object and call the GasPumpedMsg strategy class function GasPumpedMsg()
payMsg ()	Create factory object and call the PayMsg strategy class function PayMsg()
printReceipt ()	Create factory object and call the PrintReceipt strategy class function PrintReceipt()
pumpGallon ()	Create factory object and call the PumpGallon strategy class function PumpGallon()
readyMsg()	Create factory object and call the ReadyMsg strategy class function ReadyMsg()
rejectMsg ()	Create factory object and call the RejectMsg strategy class function RejectMsg()
returnCash ()	Create factory object and call the ReturnCash strategy class function ReturnCash()
setinitialValues ()	Create factory object and call the SetInitialMsg strategy class function SetInitialValues()
setPrice ()	Create factory object and call the SetPrice strategy class function SetPrice()
stopMsg ()	Create factory object and call the StopMsg strategy class function StopMsg()
storeCash ()	Create factory object and call the StoreCash strategy class function StoreCash()
storePrices ()	Create factory object and call the StoreData strategy class function StorePrices()
enterPinMsg()	Create factory object and call the EnterPinMsg() strategy class function EnterPinMsg()
WrongPinMsg()	Create factory object and call the WrongPinMsg strategy class function WrongPinMsg()
InitializeData()	Create factory object and call the InitializeData strategy class function InitializeData()

Data Store

Class DataStore:

Purpose and Attributes - Abstract class for DataStore; classes DataStore1 and DataStore2 extend this class DataStore.

Class DataStore1:

Purpose and Attributes	Represents the concrete class for DataStore in GasPump1
Variables	Permanent: gasType as String Rprice as float Sprice as float Price as float G as int // number of gallons Total as float Pin as String Temp_p as String Temporary: a as float // *a* is the price of the Regular gas per gallon b as float // *b* is the price of Super gas per gallon

Class DataStore2:

Purpose and Attributes	Represents the concrete class for DataStore in GasPump2
Variables	Permanent: gasType as String Rprice as int Sprice as int Pprice as int Cash as float Price as int L as int // L is number of liters Total as int M as int Temporary: a as int // *a* is the price of Regular gas, b as int // *b* is the price of Premium gas c as int // *c* is the price of Super gas per liter temp_cash as float.

4. Sequence Diagram

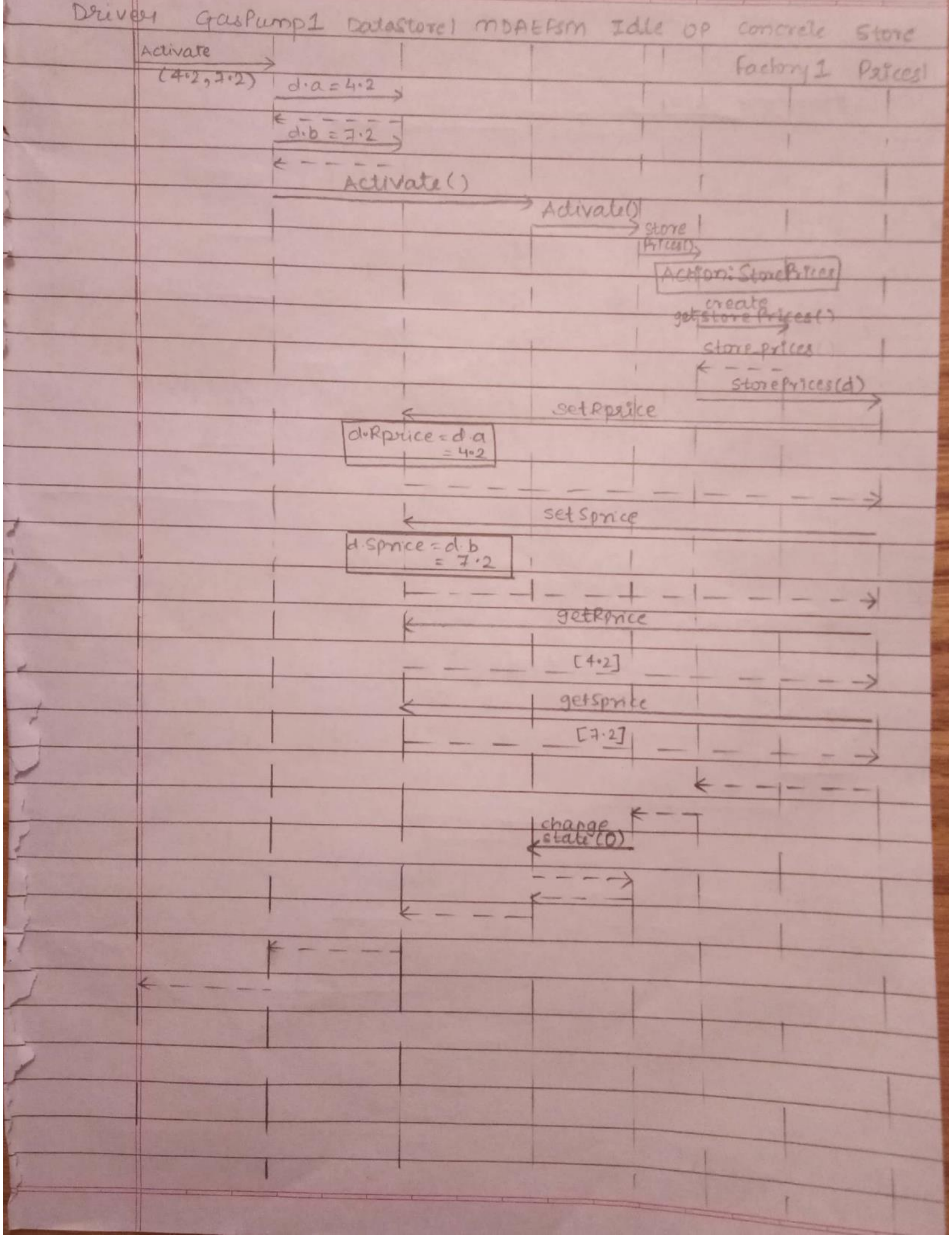
i. Scenario-I should show how one gallon of Diesel gas is disposed in GasPump-1, i.e., the following sequence of operations is issued: Activate(4.2, 7.2), Start(), PayDebit(“abc”), Pin(“abc”), Diesel(), StartPump(), PumpGallon(), FullTank()

Scenario 1

PAGE No. 1

DATE / /

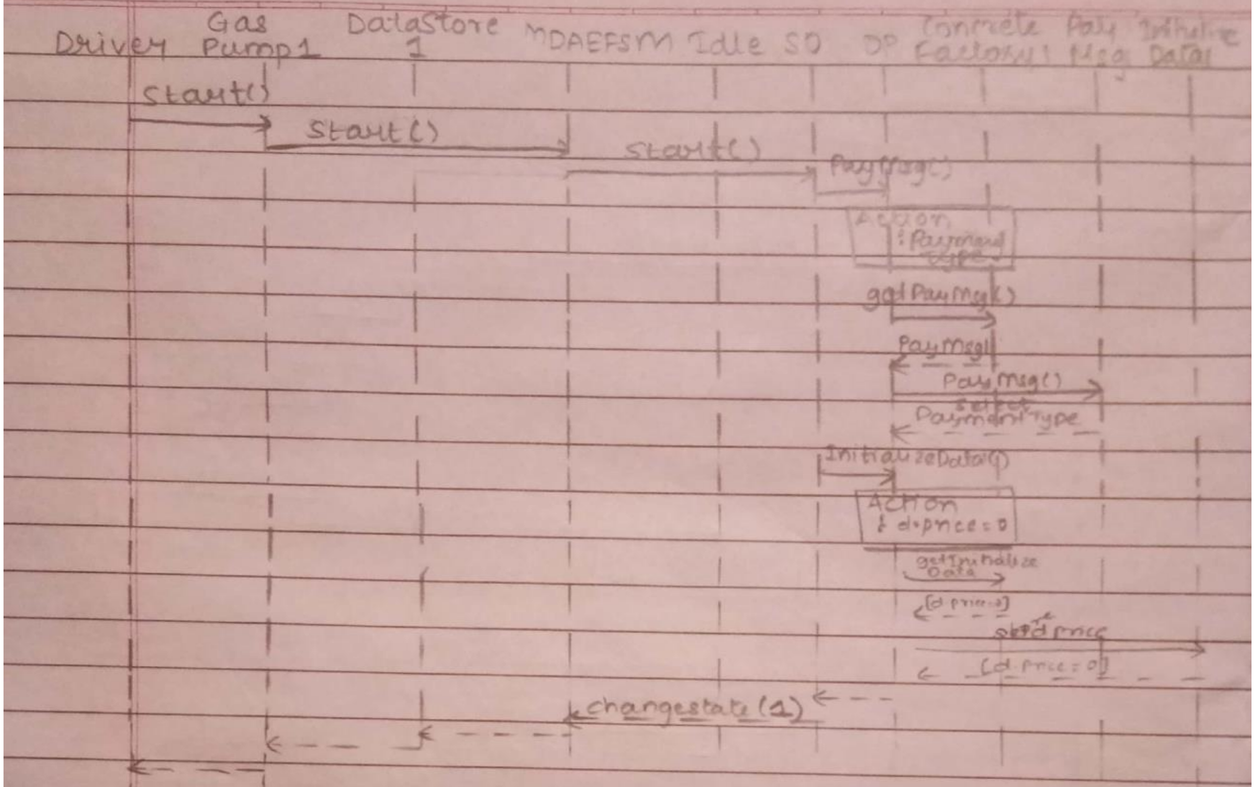
Action 1.1 Activate (4.2, 7.2)



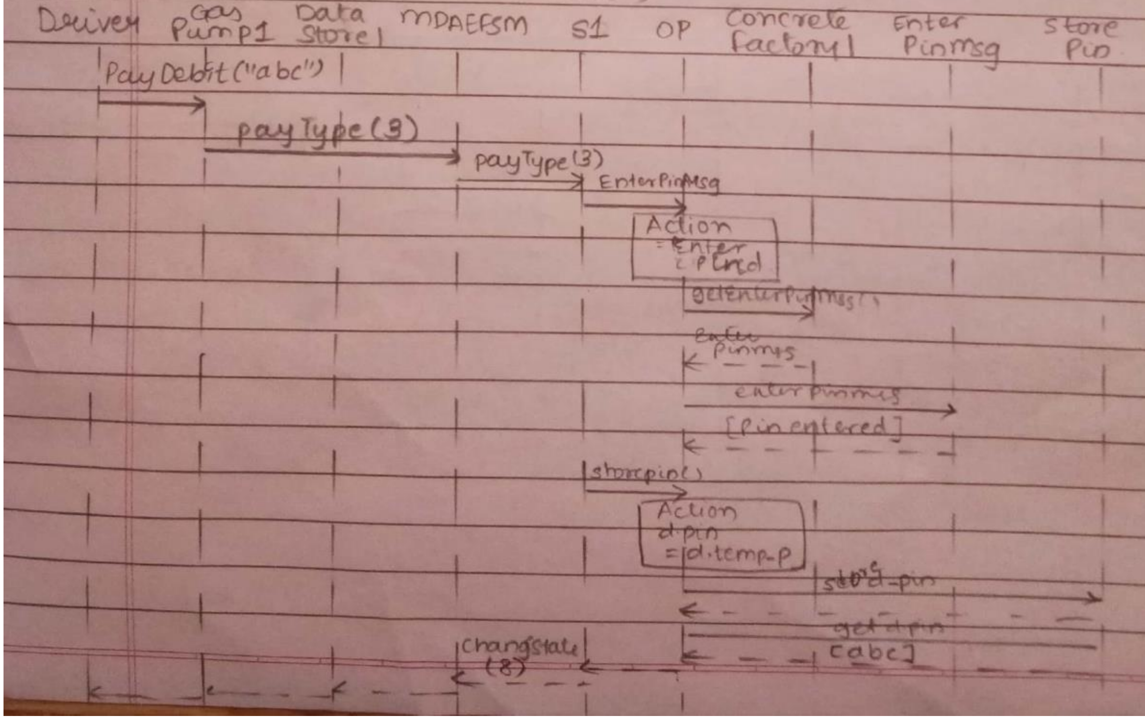
PAGE NO

DATE / /

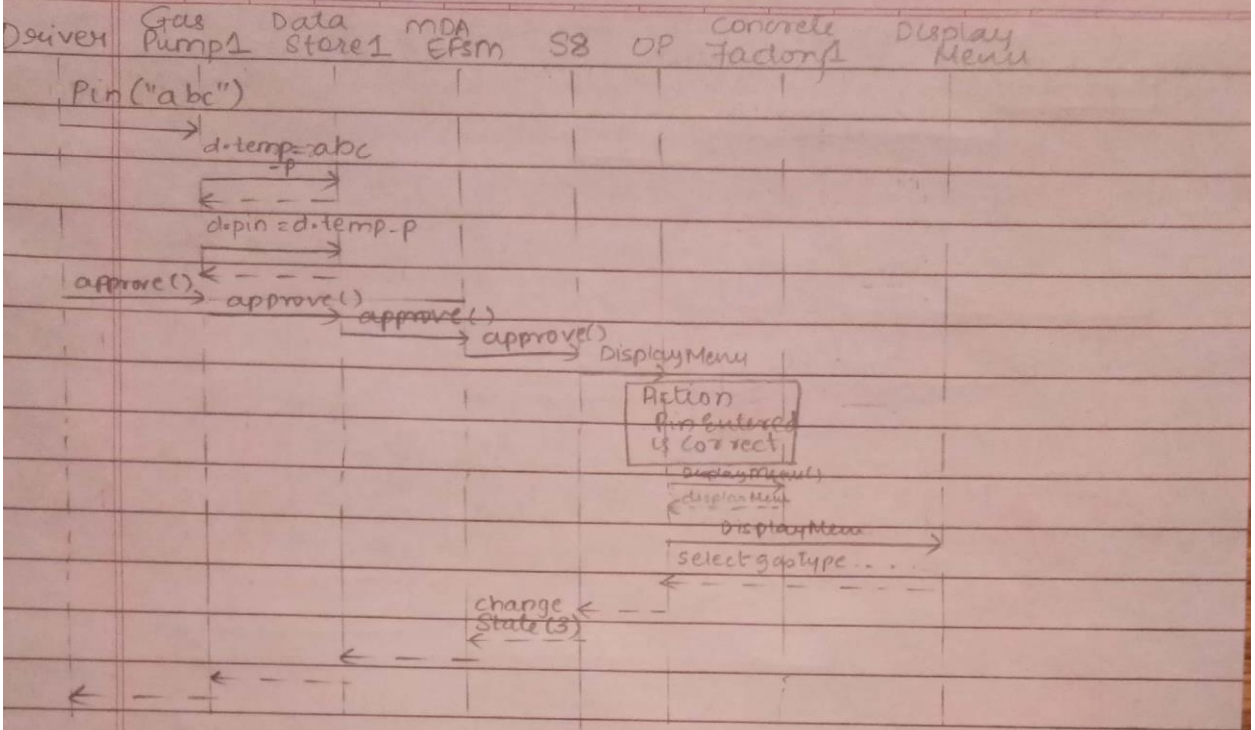
Action Start()



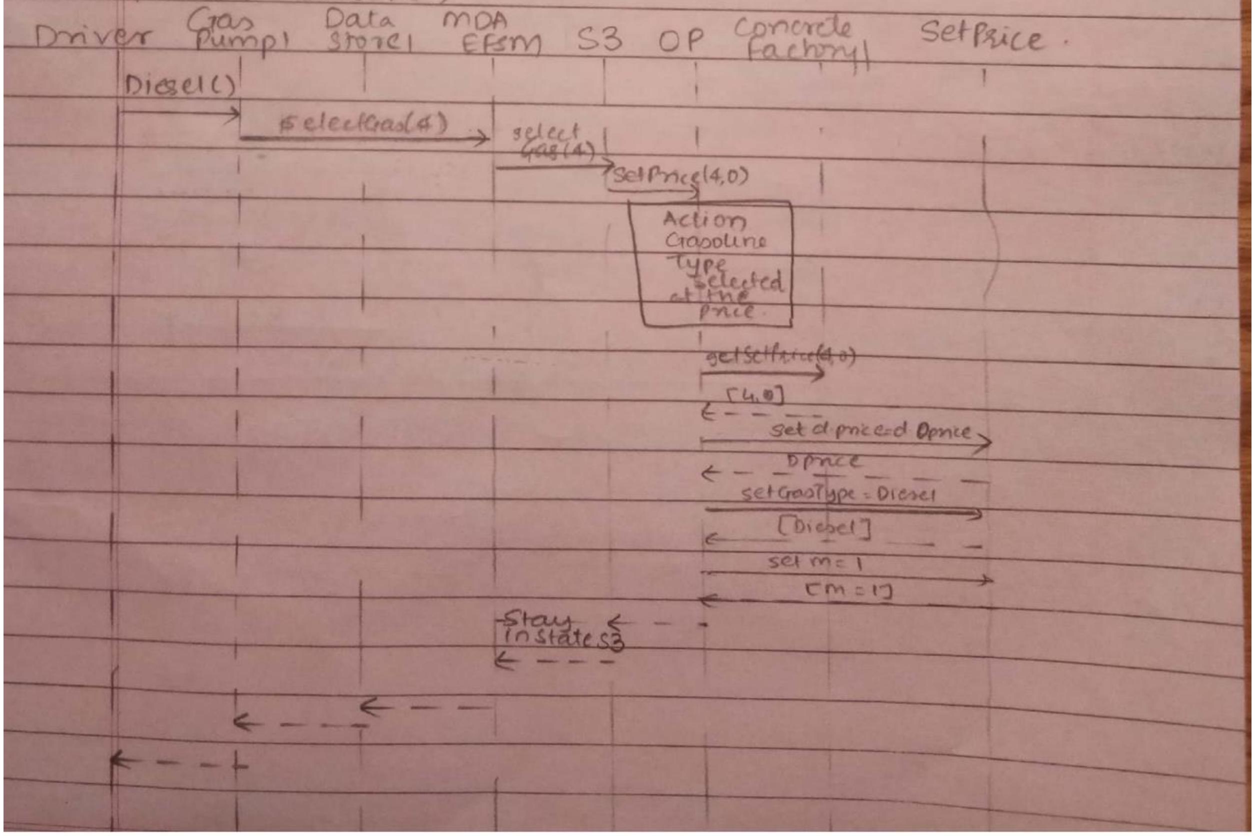
Action PayDebit("abc")



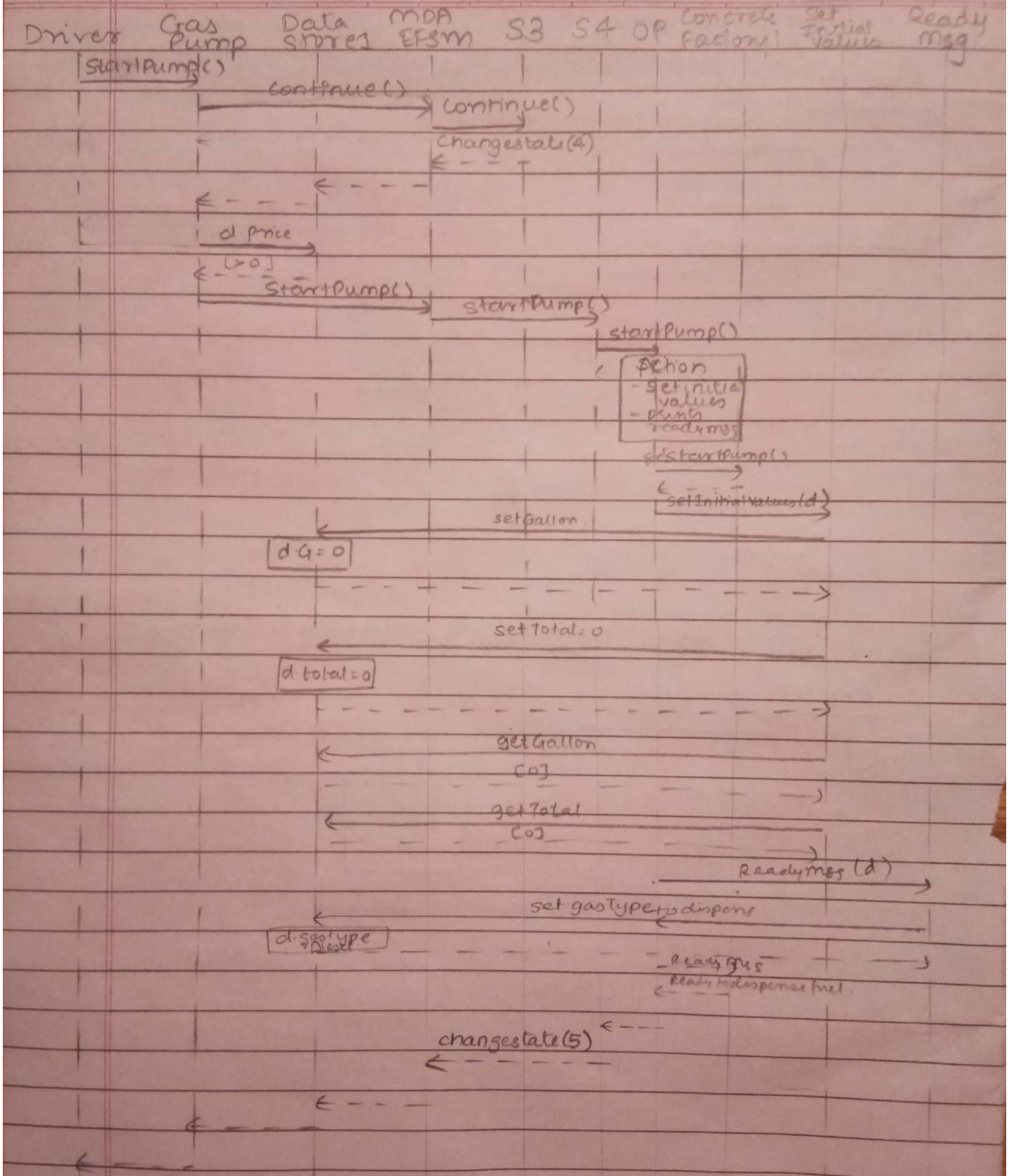
Action Pin("abc")



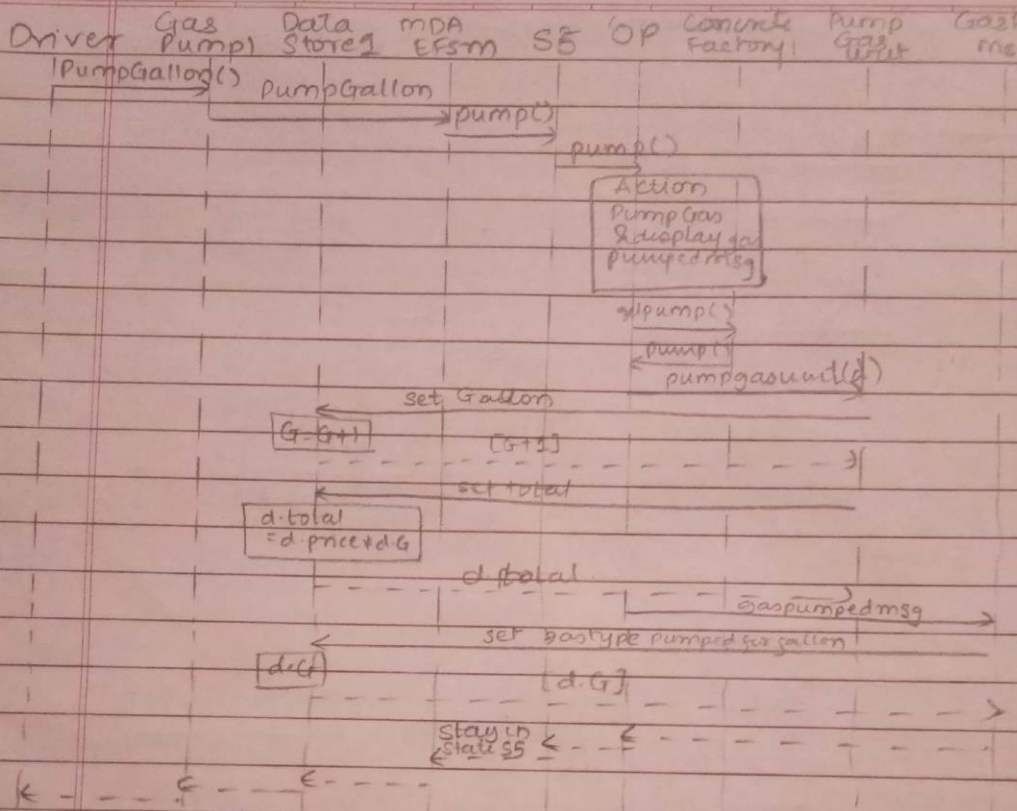
Action Diesel( )



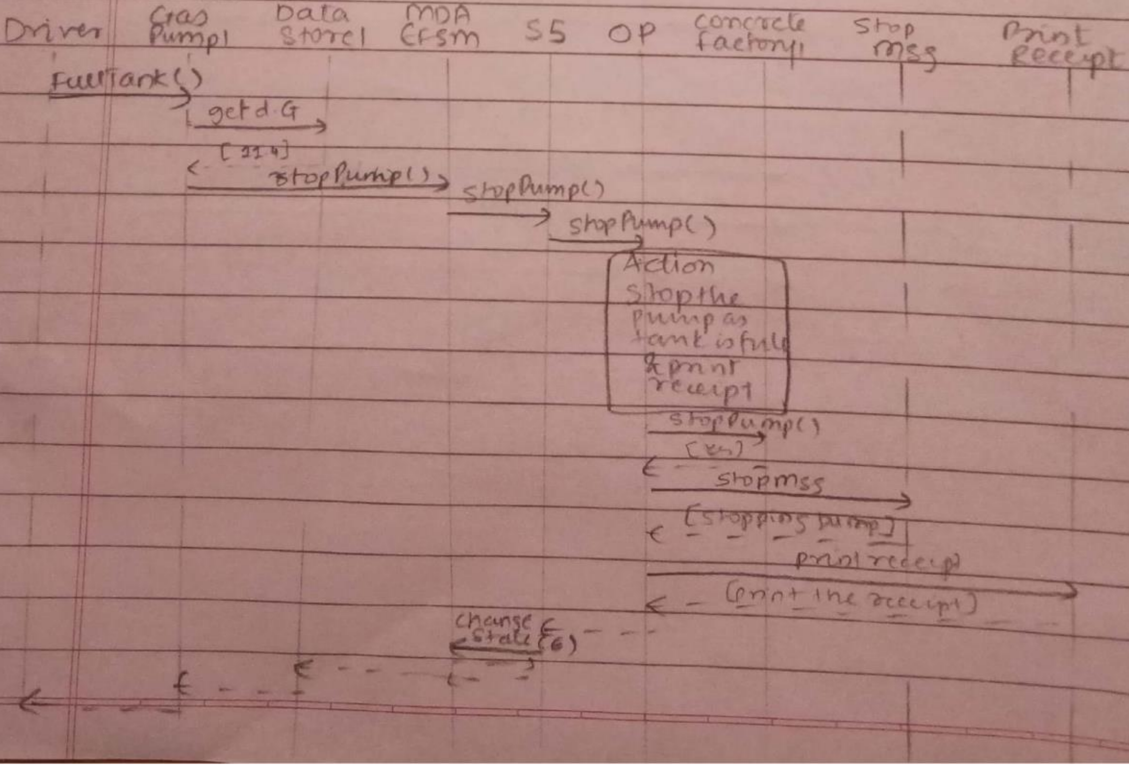
StarlPump()


Driver
Gas Pump
Data store1
MDA EFSM
S3
S4
OP
Concrete Factory1
Set Initial Values
Ready msg
StartPump()
continue()
continue()
Changestate(4)
d.price
[>0]
StartPump()
startPump()
startPump()
Action
- set initial values
- prints readymsg
d.startPump()
setInitialValues(d)
setGallon
d G=0
setTotal=0
d total=0
getGallon
[0]
getTotal
[0]
Readymsg (d)
set gasTypetodispens
d.gastype Diesel
ReadyMsg
Ready todispense fuel.
changestate(5)

PumpGallon()



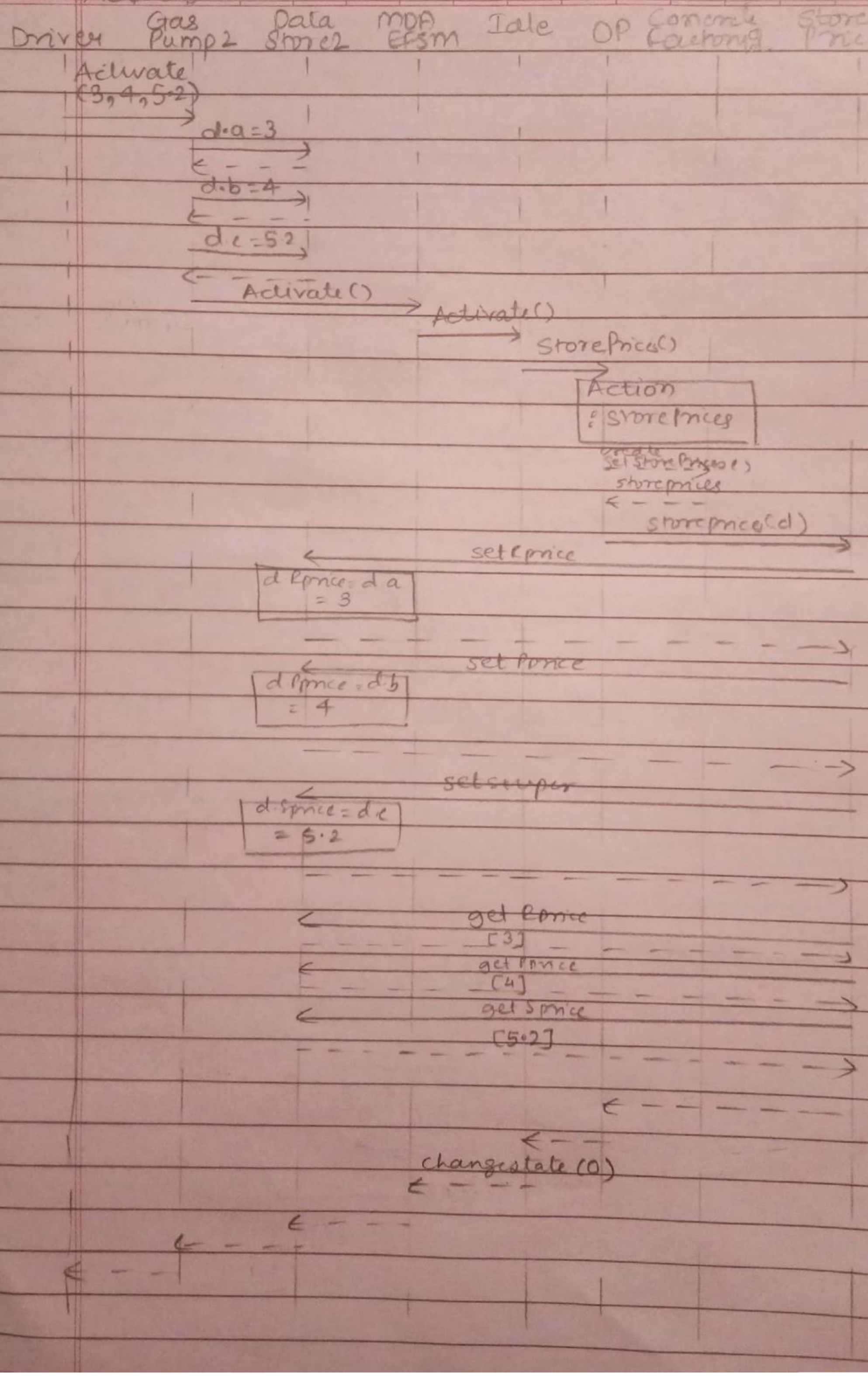
FullTank



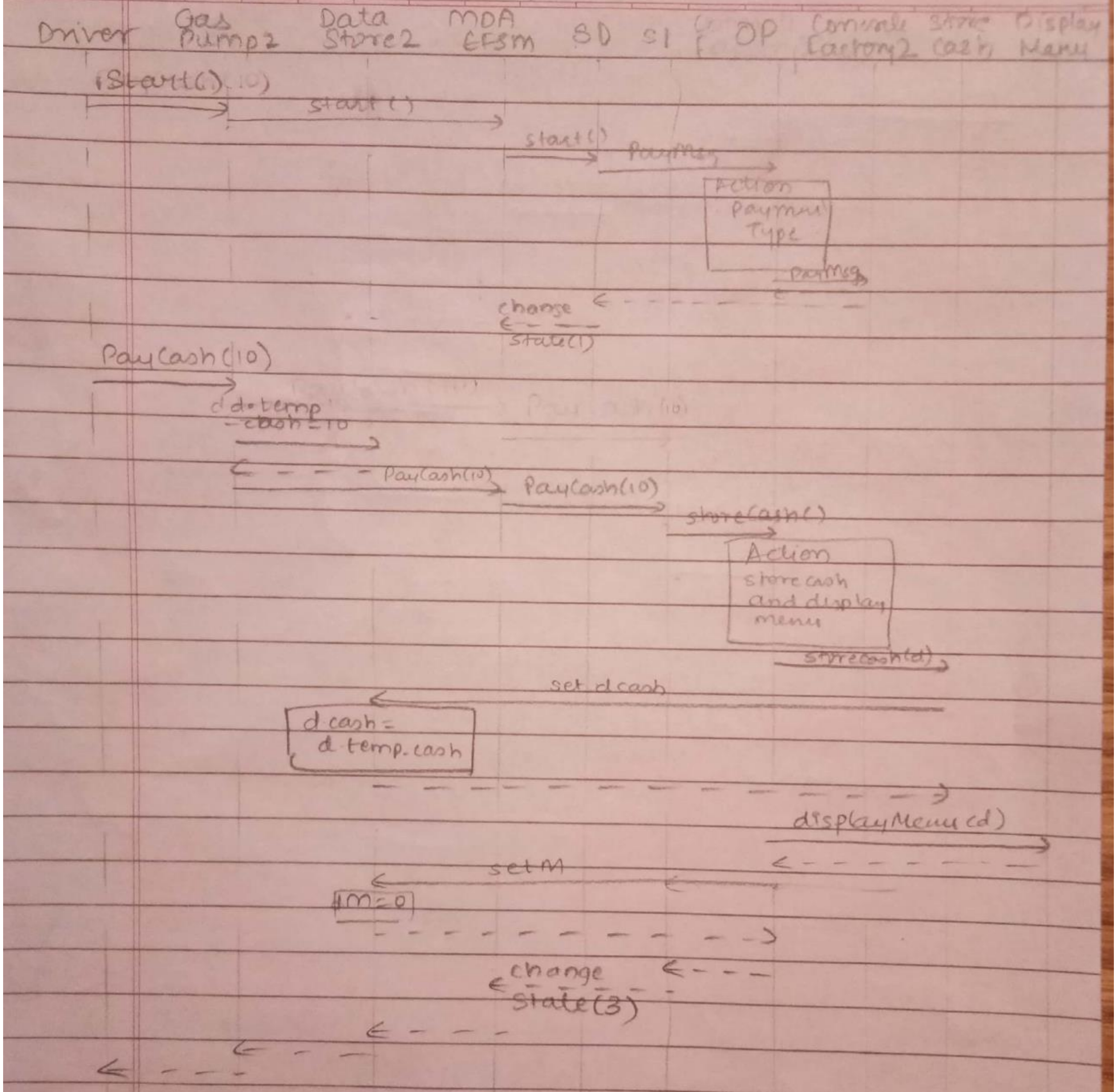
ii. Scenario-II should show how one liter of Premium gas is disposed in GasPump-2, i.e., the following sequence of operations is issued: Activate(3, 4, 5.2), PayCash(10), Premium(), StartPump(), PumpLiter(), PumpLiter(), NoReceipt()

Scenario 2

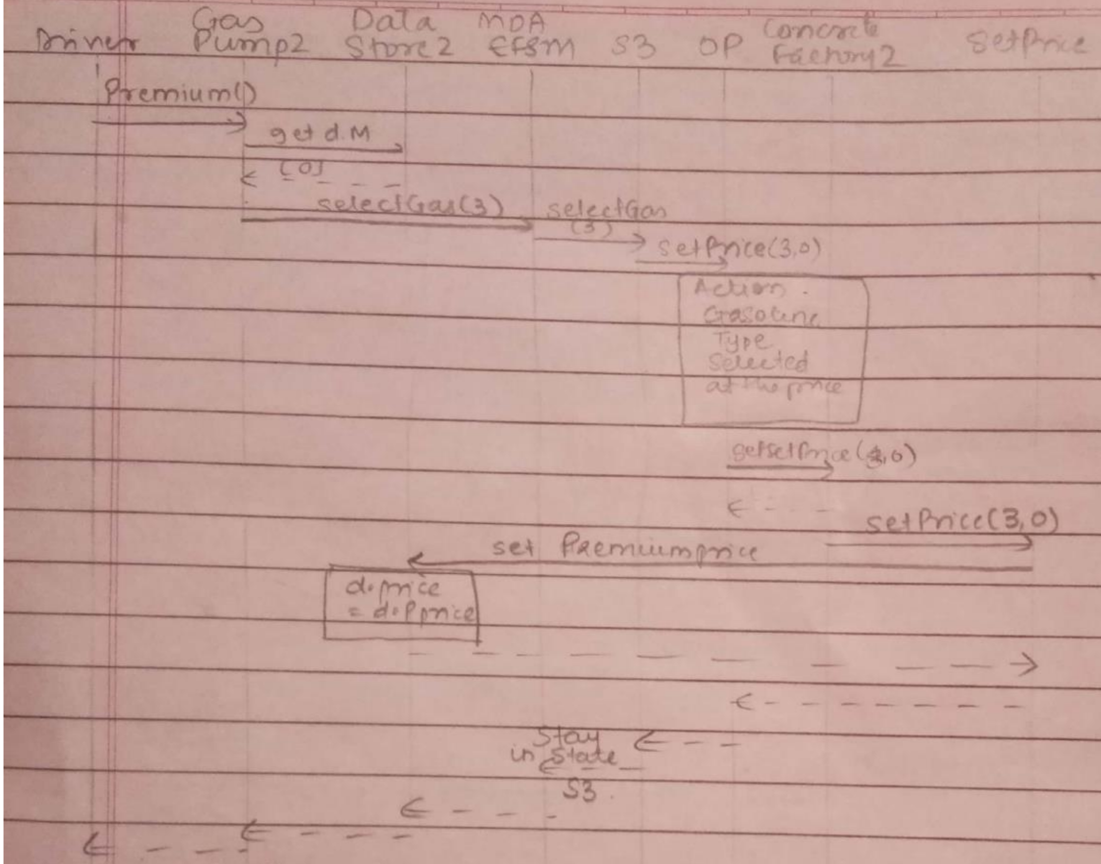
Action - Activate (3, 4, 5.2)



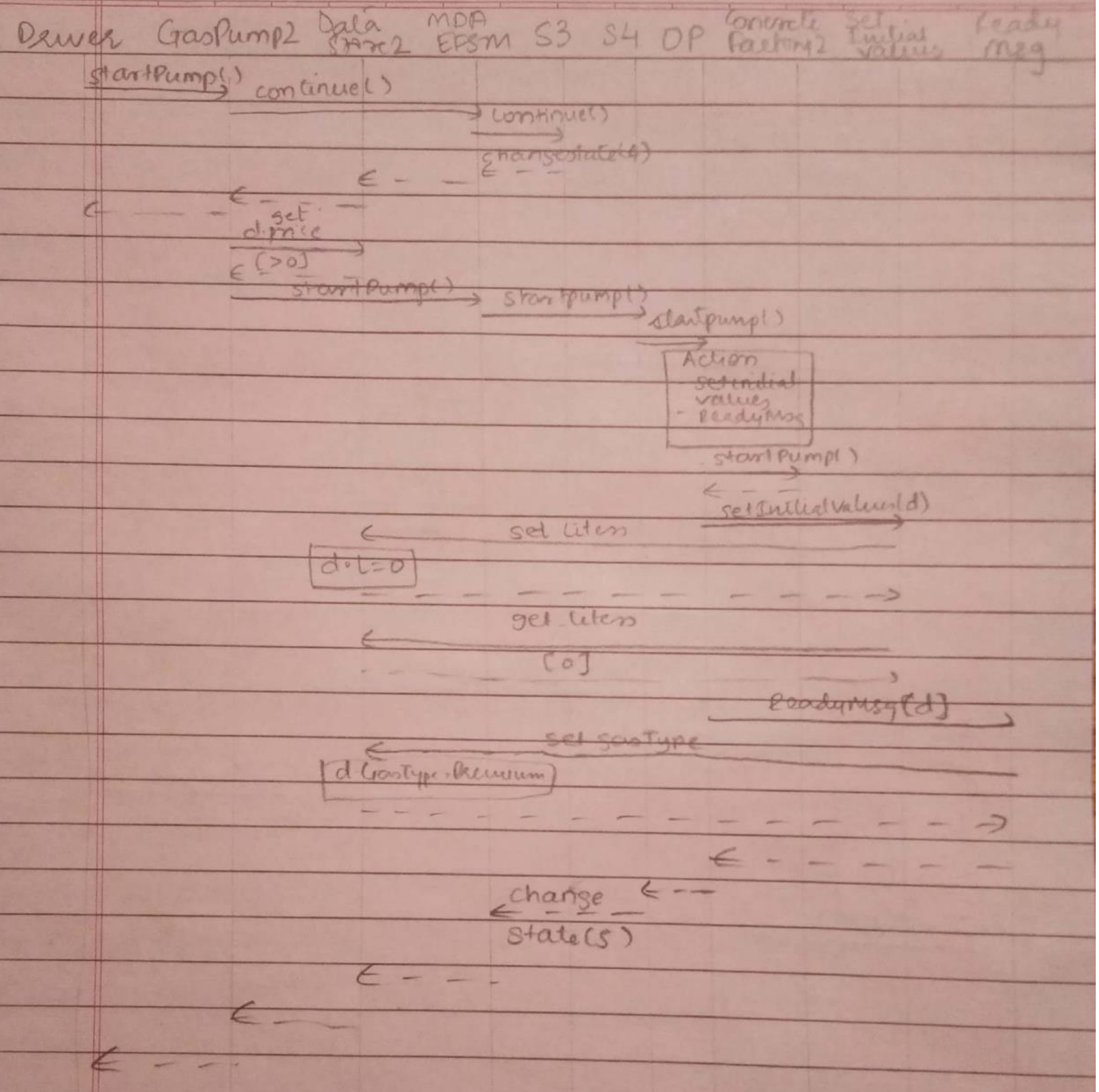
Action Paycash(10)



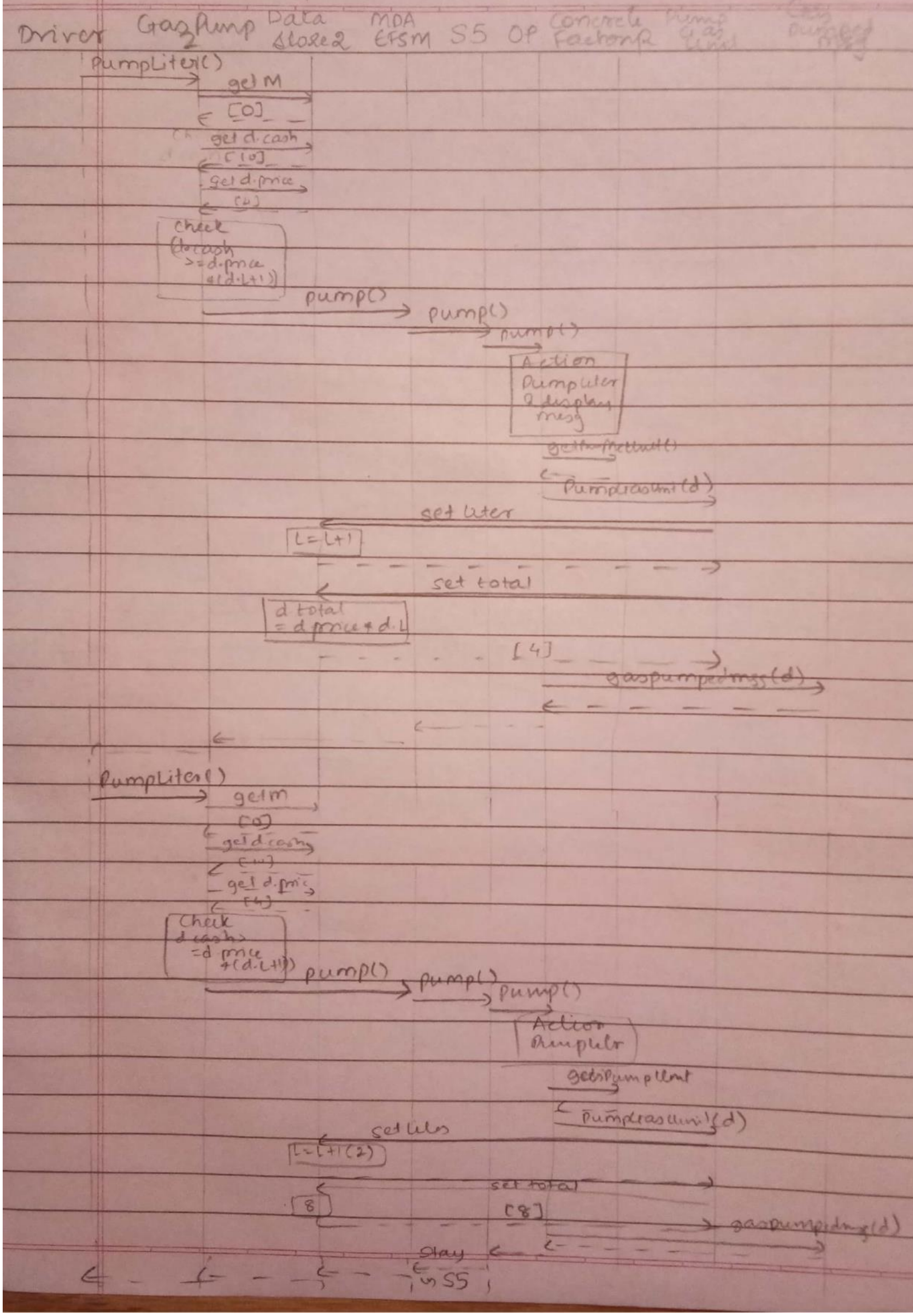
Action & Premium



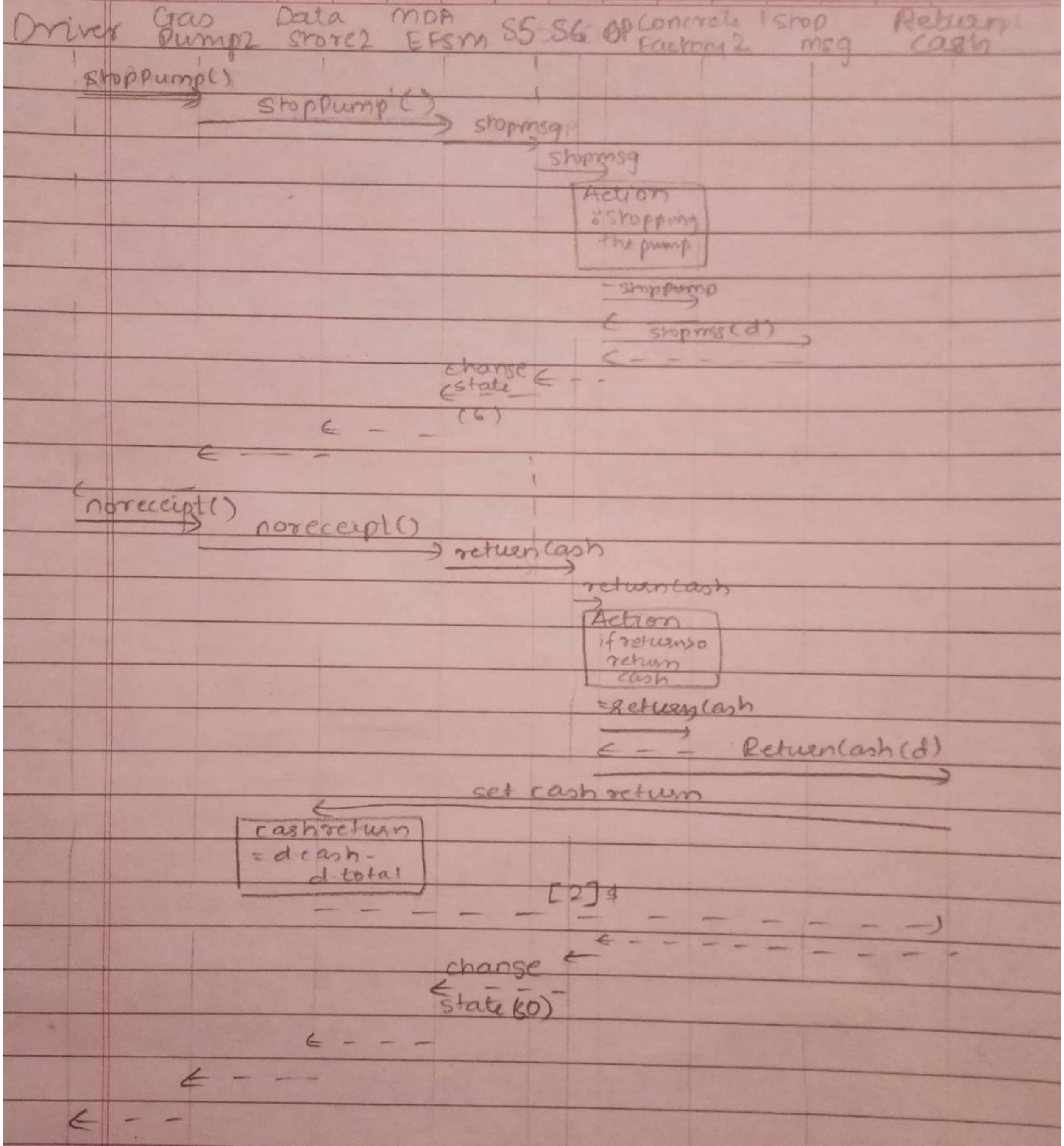
Action startPump()



Action Pump Liter()



Action NoReceipt()



5. Source Code

5.1 Driver

```
package Main;

import AbstractFactory.*;
import GasPump.*;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.Scanner;

public class Driver {
    public static void main(String[] args) {
        BufferedReader scan = new BufferedReader(new InputStreamReader(System.in));

        System.out.println("Select the GasPump:");
        System.out.println("1.GasPump1");
        System.out.println("2.GasPump2");

        int pump_type;
        String input = "initial";
        String x = null;
        String p = null;
        try {
            pump_type = Integer.parseInt(scan.readLine());
            switch (pump_type) {
                case 1: {
                    ConcreteFactory1 cf1 = new ConcreteFactory1();
                    GasPump1 gp1 = new GasPump1(cf1);
                    System.out.println(
                            "GasPump-1 " +
                                    "\n Operations to Perform: " + "\n 0. Activate(float a, float b) " + "\n 1.
Start " + "\n 2. PayDebit " + "\n 3. For Debit(Pin) " +"\n 4. ApproveDebit" + "\n c. PayCredit "
+"\n a. ApproveCredit " +"\n r. Reject "
                                    + "\n 5. Regular " + "\n 6. Diesel " + "\n 7. Cancel " + "\n 8. StartPump " +
                                    "\n 9. PumpGallon " +"\n 10. FullTank " + "\n s. StopPump" + "\n e. Exit"
                    );

                    while (!input.equals("e")) {
                        gp1.printOperations();
                        input = scan.readLine();
                        switch (input) {
                            case "0":
                            {   // Activate function
                                System.out.println("GasPump has been activated");
    float a, b;
    System.out.println("Enter the price for the Regular Gas: ");
    try
    {
        a = Float.parseFloat(scan.readLine());
        System.out.println("Enter the price for the Diesel Gas: ");
        b = Float.parseFloat(scan.readLine());
        gp1.Activate(a, b);
    }
    catch (NumberFormatException e)
    {
        System.out.println("Please enter a valid number --> float");
    }
    break;
}
case "1": { // Start function
    System.out.println(" GasPump has been started");
    gp1.Start();
    break;
}
case "2": { // PayDebit function
    System.out.println("PayDebit for any purchase");
    break;
}
case "3": { // Pin function
    System.out.println("Enter the pin: ");
    Scanner in = new Scanner(System.in);
    x = in.nextLine();
    gp1.Pin(x);
    gp1.PayDebit();
    break;
}
case "4": { // Approve function
    System.out.println("Please enter the pin: ");
    Scanner in = new Scanner(System.in);
    p = in.nextLine();
    gp1.approve(p);
    break;
}
case "r": { // Reject function
    System.out.println("PayType has been rejected");
    gp1.Reject();
    break;
}
case "5": { // Regular gas selected
    System.out.println("Regular gas has been selected");
```

```
   gp1.Regular();
   break;
}
case "6": { // Diesel gas selected
   System.out.println("Diesel gas has been selected");
   gp1.Diesel();
   break;
}
case "7": { // Cancel function
   System.out.println("Process Cancelled");
   gp1.Cancel();
   break;
}
case "8": { // StartPump function
   System.out.println("GasPump has started to dispense the gas");
   gp1.StartPump();
   break;
}
case "9": { // PumpGallon function
   System.out.println("Gas Dispensed");
   gp1.PumpGallon();
   break;
}
case "c":{ //PayCredit function
   System.out.println("PayCredit for any purchase");
   gp1.PayCredit();
   break;
}
case "a":{ //Approve credit card function
   gp1.approvec();
   break;
}
case "10": { // FullTank
   System.out.println("Tank Full");
   gp1.FullTank();
   break;
}
case "s": { // StopPump
   System.out.println("GasPump has been stopped");
   gp1.StopPump();
   break;
}
case "e": { // Exit
   break;
}
default: { // selected operation other than the ones mentioned in menu
```

```
                                System.out.println("Make a valid selection '" + input + "'");
                                break;
                            }
                        }
                    } // end of while
                    System.out.println("Exiting from GasPump1");
                    break;
                } // end of GasPump1
                case 2: {
                    ConcreteFactory2 cf2 = new ConcreteFactory2();
                    GasPump2 gp2 = new GasPump2(cf2);
                    System.out.println(
                            "GasPump-2 " +
                                    "\n Operations to Perform: " + "\n 0. Activate (int a, int b, int c)" + "\n 1.
PayCash" + "\n 2. PayCredit" +
                                    "\n a. Approve " + "\n r. Reject " + "\n 3. Regular " + "\n 4. Super " + "\n
5. Premium " + "\n 6. Cancel" + "\n 7. StartPump" +
                                    "\n 8. PumpLiter" + "\n 9. Stop " + "\n p. PrintReceipt" + "\n n.
NoReceipt" + "\n e. Exit");

                    while (!input.equals("e")) {
                        gp2.printOperations();
                        input = scan.readLine();
                        switch (input) {
                            case "0": { // Activate function
                                System.out.println("GasPump has been activated");
                                int a, b, c;
                                System.out.println("Enter the price for the Regular Gas ");
                                try {
                                    a = Integer.parseInt(scan.readLine());
                                    System.out.println("Enter the price for the Premium Gas ");
                                    b = Integer.parseInt(scan.readLine());
                                    System.out.println("Enter the price for the Super Gas ");
                                    c = Integer.parseInt(scan.readLine());
                                    gp2.Activate(a, b, c);
                                } catch (NumberFormatException e) {
                                    System.out.println("Please enter a valid number --> int");
                                }
                                break;
                            }
                            case "1": { // PayCash function
                               gp2.Start();
                                System.out.println("Insert cash amount");
                                try {
                                    float cash = Float.parseFloat(scan.readLine());
                                    gp2.PayCash(cash);
```

```
    } catch (NumberFormatException e) {
        System.out.println("Please enter a valid number --> float");
    }
    break;
}
case "2":{ // PayCredit function
    gp2.Start();
    System.out.println("PayCredit for any purchase");
    gp2.PayCredit();
    break;
}
case "a":{ // Approve the credit card function
    gp2.approvec();
    break;
}
case "r": { // Reject function
    System.out.println("PayType has been rejected");
    gp2.Reject();
    break;
}
case "3": { // Regular gas selected
    System.out.println("Regular gas has been selected");
    gp2.Regular();
    break;
}
case "4": { // Super gas selected
    System.out.println("Super gas has been selected");
    gp2.Super();
    break;
}
case "5": { // Premium gas selected
    System.out.println("Premium gas has been selected");
    gp2.Premium();
    break;
}
case "6": { // Cancel function
    System.out.println("Process has been cancelled");
    gp2.Cancel();
    break;
}
case "7": { // Start function
    System.out.println("GasPump has been started");
    gp2.StartPump();
    break;
}
case "8": { // PumpLiter function
```

```
                                System.out.println("Gas has been dispensed");
                                gp2.PumpLiter();
                                break;
                            }
                            case "9": { // Stop function
                                System.out.println("GasPump has been stopped");
                                gp2.Stop();
                                break;
                            }
                            case "p": { // PrintReceipt
                                System.out.println("Printing the Receipt");
                                gp2.Receipt();
                                break;
                            }
                            case "n": { // NoReceipt
                                System.out.println("No Receipt has been selected");
                                gp2.NoReceipt();
                                break;
                            }
                            case "e": { // Exit
                                break;
                            }
                            default: { //selected operation other than the ones mentioned in menu
                                    System.out.println("Make a valid selection '" + input + "'");
                                break;
                            }
                        }
                    } // end of while
                    System.out.println("Exiting from GasPump2");
                    break;
                } // end of GasPump2
                default: {
                    System.out.println("Select a valid GasPump");
                    System.exit(1);
                }

            }
        } catch (IOException ioe) {
            System.out.println("Terminating Application");
            System.exit(1);
        }
    }
}
```

5.2 Gas Pump

```
package GasPump;

import AbstractFactory.AbstractFactory;
import DataRepository.DataStore;
import MDA.MDAEFSM;
import Output.Output;
/*
 *
 * This class is inherited by GasPump1 and GasPump2 and they will call the superclass's
constructor passing in its own ConcreteFactory
    as the AbstractFactory field. The ConcreteFactory class makes sure each returned object has
the proper
    object references
*/
public abstract class GasPump {
    DataStore data;
    MDAEFSM model;
    GasPump(AbstractFactory af) {
        this.data = af.getDataObj();
        this.model = new MDAEFSM();
        this.model.setOP(new Output(af));
    }
    /*
        display menu corresponding to each GasPump
     */
    public abstract void printOperations();

}
```

5.2.1 GasPump1

```
package GasPump;
import AbstractFactory.AbstractFactory;
import DataRepository.DataStore1;
/*
    Processing input of GasPump1
 */
public class GasPump1 extends GasPump {
    public GasPump1(AbstractFactory af) {
        super(af);
    }
    /*
        display menu
     */
    @Override
    public void printOperations() {
        System.out.println(
                        "\nSelect operation: " +
              "\n 0. Activate(float a, float b) " + "   1. Start " + "   2. PayDebit " + "   3. For
  Debit(Pin) "+"    4.ApproveDebit" +
              "\n c. PayCredit " +"    a. ApproveCredit " + " r. Reject "+
              "\n 5. Regular " + "     6. Diesel " +
              "\n 7. Cancel " + "    8. StartPump " + "    9. PumpGallon " +
              "\n 10. FullTank " + "   s. StopPump " + "   e. Exit "
      );
    }
    /*
      Check the input parameters and call the
      activate()
      a: price of Regular gas
      b: price of Diesel gas
     */
    public void Activate(float a, float b) {
      if (a > 0 && b > 0) {
        DataStore1 d = (DataStore1) this.data;
        d.a = a;
        d.b = b;
        model.activate();
      }
      else {
        System.out.println("Failed! Price should be > $0");
      }
    }
    /*
      Call the start()
    */
    public void Start() {
      model.start();
    }
    /*
      Call the payType() and print a debit card authentication message
     */
    public void PayDebit() {
      model.payType(3);
    }
    /*
      Call the payType() and print a credit card authentication message
     */
    public void PayCredit() {
      model.payType(1);
    }
    /*
      Call the Pin() with x-input as the string
    */
```

```
public void Pin(String x) {
  DataStore1 d = (DataStore1) data;
  d.temp_p=x;
  d.pin=d.temp_p;
}
/*
   Call the reject()
 */
public void Reject() {
   model.reject();
}
/*
   Call the cancel()
 */
public void Cancel() {
   model.cancel();
}
/*
   Call the selectGas()and pass in 1 as the gas-type
 */
public void Regular() {
   model.selectGas(1,1);
}
/*
   Call the selectGas()and pass in 4 as the gas-type
 */
public void Diesel() {
   model.selectGas(4,1);
}
/*
   Call the startPump()
 */
public void StartPump() {
   DataStore1 d = (DataStore1) data;
   model.Continue();
   if(d.price>0)
   {
   model.startPump();
   }
}
/*
   Call the pump()
 */
public void PumpGallon() {
   model.pump();
}
```

```
    /*
        Call the approve() for debit card
     */
    public void approve(String p)
    {
       DataStore1 d = (DataStore1) this.data;
       if(p.equals(d.pin))
       {
           model.correctPin();
       }
       else
       {
           model.incorrectPin();
       }
    }
    /*
        Call the approvec() for credit card
     */
    public void approvec()
    {
        model.approvec();
    }
    /*
        call the stopPump()
        GasPump1 always prints receipts.
     */
    public void StopPump() {
        model.stopPump();
    }
    /*
    call the FullTank() and receipt()
    */
     public void FullTank() {
        model.receipt();
    }
}
```

5.2.2 GasPump2

```
package GasPump;
import AbstractFactory.AbstractFactory;
import DataRepository.DataStore2;
/*
    Processor input for GasPump2
 */
public class GasPump2 extends GasPump {
    public GasPump2(AbstractFactory af) {
        super(af);
    }
    /*
        display Menu
     */
    @Override
    public void printOperations() {
        System.out.println(
                        "\nSelect operation: " +
                        "\n 0. Activate(int a, int b, int c)" + "    1. PayCash " +
                        "\n 2. PayCredit "+ "    a. Approve " + "    r. Reject " +
                        "\n 3. Regular " + "    4. Super " + "    5. Premium " + "    6. Cancel " +
                        "\n 7. StartPump " + "    8. PumpLiter " + "    9. Stop " +
                        "\n p. PrintReceipt " + "     n. NoReceipt " + "    e. Quit the program ");
     }
    /*
        Check the input parameters and call activate()

        a: price of Regular gas
        b: price of Super gas
        c: price of Premium gas
     */
    public void Activate(int a, int b, int c) {
        if (a > 0 && b > 0 && c > 0) {
            DataStore2 d = (DataStore2) data;
            d.a = a;
            d.b = b;
            d.c = c;
            model.activate();
        } else {
            System.out.println("Failed! Price should be > $0");
        }
    }
    /*
        Call the start()
     */
   public void Start() {
        model.start();
    }
    /*


        call the payType()
     */
    public void PayCash(float cash) {
        if (cash > 0) {
            DataStore2 d = (DataStore2) data;
            d.temp_cash = cash;
        model.payType(2);
        d.M=0;

    } else {
        System.out.println("Cash should be > $0");
    }
}
/*
    Call the cancel()
 */
public void Cancel() {
    model.cancel();
}
/*
    Call the Reject()
 */
public void Reject() {
    model.reject();
}
/*
    Call the selectGas()and pass in 1 as the gas-type
 */
public void Regular() {
    DataStore2 d = (DataStore2) data;
    if(d.M==1)
    {
    model.selectGas(1,1);
    }
    else
    model.selectGas(1, 0);
}
/*Call the selectGas()and pass in 2 as the gas-type
 */
public void Super() {
   DataStore2 d = (DataStore2) data;
    if(d.M==1)
    {
    model.selectGas(2,1);
    }
    else
    model.selectGas(2, 0);
}
/*Call the selectGas()and pass in 3 as the gas-type
 */
public void Premium() {
   DataStore2 d = (DataStore2) data;
```

```
    if(d.M==1)
    {
    model.selectGas(3,1);
    }
    else
    model.selectGas(3, 0);
}
/*
    Call the startPump()
 */
public void StartPump() {
    DataStore2 d = (DataStore2) data;
    model.Continue();
    if(d.M==1)
    {
        model.startPump();
        d.price= (float)(1.1 * d.price);
        d.L=0;
        //System.out.println("Price*****="+d.price+"Liters="+d.L);
    }
    else if(d.M==0)
    {
        model.startPump();
        d.L=0;
        //System.out.println("Price="+d.price+"Liters="+d.L);
    }
}
public void PayCredit() {
    DataStore2 d = (DataStore2) data;
        model.payType(1);
        d.M=1;
    }
/*
    If there is not enough cash to pump another liter, print a message indicating as such,
    and call the stopPump()

    Otherwise, call the pump()
 */
public void PumpLiter() {
    DataStore2 d = (DataStore2) data;
    if ((d.M==0) && (d.cash < d.price * (d.L + 1)))
    {
        System.out.println("Limited Balance");
        model.stopPump();
    }
    else if((d.M==0) && (d.cash>=d.price * (d.L + 1)))
```

```
        {
            model.pump();

        }
        else if((d.M==1))
        {
            model.pump();
        }
    }
    /*
        Call the approvec() for credit card
     */
    public void approvec()
    {
        model.approvec();
    }
    /*
        Call the approve() for debit card
     */
    public void approve()
    {
       DataStore2 d = (DataStore2) data;
        d.M=1;
    }
    /*
        Call the stopPump()
     */
    public void Stop() {
        model.stopPump();
    }
    /*
        Call the receipt()
     */
    public void Receipt() {
        model.receipt();
    }
    /*
        Call the noReceipt()
     */
    public void NoReceipt() {
        model.noReceipt();
    }
}
```

5.3 Output

```
package Output;

import AbstractFactory.AbstractFactory;
import DataRepository.DataStore;
import Strategy.CancelMsg;
import Strategy.DisplayMenu;
import Strategy.GasPumpedMsg;
import Strategy.PayMsg;
import Strategy.PrintReceipt;
import Strategy.PumpGasUnit;
import Strategy.ReadyMsg;
import Strategy.RejectMsg;
import Strategy.ReturnCash;
import Strategy.SetInitialValues;
import Strategy.SetPrice;
import Strategy.StopMsg;
import Strategy.StoreCash;
import Strategy.StorePrices;
import Strategy.EnterPinMsg;
import Strategy.InitializeData;
import Strategy.StorePin;
import Strategy.WrongPinMsg;
/*
    Output processor for the Gas Pump

 */
public class Output {
    private CancelMsg cancelMsg;
    private DisplayMenu displayMenu;
    private GasPumpedMsg gasPumpedMsg;
    private PayMsg payMsg;
    private PrintReceipt printReceipt;
    private PumpGasUnit pumpGasUnit;
    private ReadyMsg readyMsg;
    private RejectMsg rejectMsg;
    private ReturnCash returnCash;
    private SetInitialValues setInitialValues;
    private SetPrice setPrice;
    private StopMsg stopMsg;
    private StoreCash storeCash;
    private StorePrices storePrices;
    private DataStore data;
    private EnterPinMsg enterPinMsg;
    private InitializeData initializeData;
    private StorePin storePin;
    private WrongPinMsg wrongPinMsg;
```

```
public Output(AbstractFactory af) {
   this.cancelMsg = af.getCancelMsg();
   this.displayMenu = af.getDisplayMenu();
   this.gasPumpedMsg = af.getGasPumpedMsg();
   this.payMsg = af.getPayMsg();
   this.printReceipt = af.getPrintReceipt();
   this.pumpGasUnit = af.getPumpGasUnit();
   this.readyMsg = af.getReadyMsg();
   this.rejectMsg = af.getRejectMsg();
   this.returnCash = af.getReturnCash();
   this.setInitialValues = af.getSetInitialValues();
   this.setPrice = af.getSetPrice();
   this.stopMsg = af.getStopMsg();
   this.storeCash = af.getStoreCash();
   this.storePrices = af.getStorePrices();
   this.storePin = af.getStorePin();
   this.enterPinMsg = af.getEnterPinMsg();
   this.initializeData = af.getInitializeData();
   this.wrongPinMsg = af.getWrongPinMsg();
}

/*
 *  Meta-actions (implemented using Strategy pattern)
 */

public void CancelMsg() {
   this.cancelMsg.cancelMsg();
}

public void DisplayMenu() {
   this.displayMenu.displayMenu();
}

public void GasPumpedMsg() {
   this.gasPumpedMsg.gasPumpedMsg();
}

public void PayMsg() {
   this.payMsg.payMsg();
}

public void PrintReceipt() {
   this.printReceipt.printReceipt();
}
```

```
public void PumpGasUnit() {
    this.pumpGasUnit.pumpGasUnit();
}

public void ReadyMsg() {
    this.readyMsg.readyMsg();
}

public void RejectMsg() {
    this.rejectMsg.rejectMsg();
}

public void ReturnCash() {
    this.returnCash.returnCash();
}

public void SetInitialValues() {
    this.setInitialValues.setInitialValues();
}

public void SetPrice(int g, int M) {
    this.setPrice.setPrice(g,M);
}

public void StopMsg() {
    this.stopMsg.stopMsg();
}

public void StoreCash() {
    this.storeCash.storeCash();
}

public void StorePin() {
    this.storePin.storePin();
}
public void StorePrices() {
    this.storePrices.storePrices();
}
public void EnterPinMsg() {
    this.enterPinMsg.enterPinMsg();
}
public void WrongPinMsg() {
    this.wrongPinMsg.wrongPinMsg();
}
public void InitializeData() {
    this.initializeData.initializeData();
    }

}
```

5.4 DataStore

```
package DataRepository;

/*
    This class groups all DataStore classes under 1 abstract superclass
*/
public abstract class DataStore {
}
```

DataStore1

```
package DataRepository;

/*
    GasPump1 data storage object for storing the Data specific to GasPump1
 */
public class DataStore1 extends DataStore {
    public String   gasType;
    public float    Rprice;
    public float    Sprice;
    public float    price;
    public int      G;
    public float    total;
    public String pin;
    public String temp_p;
     // temporary variables
    public float a;
    public float b;

}
```

DataStore2

```
package DataRepository;

/*
GasPump2 data storage object for storing the Data specific to GasPump2
 */
public class DataStore2 extends DataStore {
    public String   gasType;
    public int      Rprice;
    public int      Sprice;
    public int      Pprice;
    public float    cash;
    public float    price;
    public int      L;
    public float    total;
    public int M;
    // temporary variables
    public int a;
    public int b;
    public int c;
    public float temp_cash;
}
```

5.5 State Pattern

MDAEFSM

```
package MDA;
import Output.Output;
/*
    State classes are responsible for performing
        1) Actions
        2) State transitions
 */

public class MDAEFSM {
    protected State s;
    protected State[] LS;
    private Output op;
    public int M;

    public MDAEFSM() {
        // list of states in the EFSM
        LS = new State[10];

        // instantiate each state, passing in a reference to this VM class
        LS[9] = new Idle(this);
        LS[0] = new S0(this);
        LS[1] = new S1(this);
        LS[2] = new S2(this);
        LS[3] = new S3(this);
        LS[4] = new S4(this);
        LS[5] = new S5(this);
        LS[6] = new S6(this);
        LS[8] = new S8(this);

        s = LS[9]; // Initially in the Idle State
    }

    public Output getOP() {
        return op;
    }
```

```
public void setOP(Output op) {
    this.op = op;
}

/*
    State operations
 */

public void activate() {
    s.activate();
}

public void start() {
    s.start();
}

/*
    t = 1 represents credit card payment type
    t = 2 represents cash payment type
    t = 3 represents debit card payment type
 */
public void payType(int t) {
    s.payType(t);
}

public void approve() {
    s.approve();
}
public void approvec()
{
    s.approvec();
    s.M=1;
}

public void reject() {
    s.reject();
}

public void cancel() {
    s.cancel();
}

/*
    g = 1 represents Regular gas
    g = 2 represents Super gas
```

```
        g = 3 represents Premium gas
        g = 4 represents Diesel gas
     */
    public void selectGas(int g, int M) {
        s.selectGas(g,M);
    }

    public void startPump() {
        s.startPump();
    }

    public void pump() {
        s.pump();
    }

    public void stopPump() {
        s.stopPump();
    }

    public void receipt() {
        s.receipt();
    }

    public void noReceipt() {
        s.noReceipt();
    }
    public void Continue()
    {
        s.Continue();
    }
    public void correctPin()
    {
        s.correctPin();
    }
    public void incorrectPin()
    {
        s.incorrectPin();
    }



}
```

STATE

```
package MDA;

/*
    State superclass in the De-centralized State Design Pattern

    most state-subclasses would only ACTUALLY implement only 1 or 2 of the methods.
    The rest would have empty bodies and that is a waste of both coding space and memory space.

 */

public abstract class State {
    MDAEFSM model;
    public int M;

    public State(MDAEFSM model) {
        this.model = model;
    }

    void activate()         {notAllowed();}
    void start()            {notAllowed();}

    /*
        credit: t=1
        cash:   t=2
        debit:  t=3
     */
    void payType(int t)     {notAllowed();}
    void approve()          {notAllowed();}
    void approvec()         {notAllowed();}
    void reject()           {notAllowed();}
    /*
        Regular:    g=1
        Super:      g=2
        Premium:    g=3
        Diesel      g=4
     */
    void selectGas(int g, int M)   {notAllowed();}
    void cancel()           {notAllowed();}
    void startPump()        {notAllowed();}
    void pump()             {notAllowed();}
    void stopPump()         {notAllowed();}
    void receipt()          {notAllowed();}
    void noReceipt()        {notAllowed();}
    void correctPin()       {notAllowed();}
    void incorrectPin()     {notAllowed();}
    void Continue() {notAllowed();}
    /*
        Print a "not allowed" message
     */
    private void notAllowed() {
        System.out.println("OPERATION NOT ALLOWED IN THIS STATE");
    }
}
```

IDLE

```
package MDA;
/*
    Idle State
 */
class Idle extends State {

    Idle(MDAEFSM model) {
        super(model);
    }

    /*
        Transit to State S0 and call the StorePrices()
     */
    @Override
    void activate() {
        if (model.s == model.LS[9]) {
            model.s = model.LS[0];
            model.getOP().StorePrices();
        }
    }
}
```

State S0

```
package MDA;

/*
    State S0
 */
class S0 extends State {

    S0(MDAEFSM model) {
        super(model);
    }

    /*
        Transit to State S1 and call the PayMsg()
     */
    @Override
    void start() {
        if (model.s == model.LS[0]) {
            model.s = model.LS[1];
            model.getOP().PayMsg();
            model.getOP().InitializeData();
            model.M=1;
        }
    }
}
```

State S1

```
package MDA;

/*
    State S1
 */
class S1 extends State {

    S1(MDAEFSM model) {
        super(model);
    }
    /*
        creditcard:
            Transition to State S2

        cash:
            Transition to State S3
            Call StoreCash() and DisplayMenu()

        debit:
            Transition to State S8
            Call EnterPinMsg() and StorePin()
     */
    @Override
    void payType(int t)
    {
        if ((t == 1) && (model.s == model.LS[1]))
        {
            model.s = model.LS[2];
            model.M=1;
        }
        else if ((t == 2) && (model.s == model.LS[1]))
        {
            model.s = model.LS[3];
            model.getOP().StoreCash();
            model.getOP().DisplayMenu();
            model.M=0;
        }
        else if((t == 3) && (model.s == model.LS[1]))
        {
            model.s = model.LS[8];
            model.getOP().EnterPinMsg();
            model.getOP().StorePin();
            model.M=1;
        }
    }
}
```

State S2

```
package MDA;

/*
    State S2
 */
class S2 extends State {

    S2(MDAEFSM model) {
        super(model);
    }

    /*
        Transition to State S3 and call DisplayMenu()
     */

    @Override
    void approvec() {
        if (model.s == model.LS[2]) {
            model.s = model.LS[3];
            System.out.println("Paytype Approved");
            model.getOP().DisplayMenu();

        }
    }
    /*
        Transition to State S2 and call RejectMsg()
     */
    @Override
    void reject() {
        if (model.s == model.LS[2]) {
            model.s = model.LS[0];
            model.getOP().RejectMsg();
        }
    }
}
```

State S3

```
package MDA;

/*
   State S3
 */
class S3 extends State {

   S3(MDAEFSM model) {
      super(model);
   }

   /*
      Stay in state S3 and call SetPrice(g,M)
    */
   @Override
   void selectGas(int g,int M) {
      if (model.s == model.LS[3]) {
         model.getOP().SetPrice(g,M);
      }
   }

   /*
      Transition to State S0 and call CancelMsg() and ReturnCash()
    */
   @Override
   void cancel() {
      if (model.s == model.LS[3]) {
         model.s = model.LS[0];
         model.getOP().CancelMsg();
         model.getOP().ReturnCash();
      }
   }

   /*
      Transition to State S4
    */
   @Override
   void Continue()
   {
      if (model.s == model.LS[3])
      {
         model.s = model.LS[4];
      }
   }
}
```

State S4

```
package MDA;

/*
    State S4
 */
class S4 extends State {

    S4(MDAEFSM model) {
        super(model);
    }

    /*
        Transition to State S5 and call SetInitialValues() and ReadyMsg()
     */
    @Override
    void startPump() {
        if ((model.s == model.LS[4]))
        {
            model.s = model.LS[5];
            model.getOP().SetInitialValues();
            model.getOP().ReadyMsg();
        }
    }
}
```

State S5

```
package MDA;
import Output.Output;
/*
    State classes are responsible for performing
        1) Actions
        2) State transitions
 */

public class MDAEFSM {
    protected State s;
    protected State[] LS;
    private Output op;
    public int M;

    public MDAEFSM() {
        // list of states in the EFSM
        LS = new State[10];

        // instantiate each state, passing in a reference to this VM class
```

```
   LS[9] = new Idle(this);
   LS[0] = new S0(this);
   LS[1] = new S1(this);
   LS[2] = new S2(this);
   LS[3] = new S3(this);
   LS[4] = new S4(this);
   LS[5] = new S5(this);
   LS[6] = new S6(this);
   LS[8] = new S8(this);

   s = LS[9]; // Initially in the Idle State
}

public Output getOP() {
   return op;
}

public void setOP(Output op) {
   this.op = op;
}

/*
   State operations
 */

public void activate() {
   s.activate();
}

public void start() {
   s.start();
}

/*
   t = 1 represents credit card payment type
   t = 2 represents cash payment type
   t = 3 represents debit card payment type
 */
public void payType(int t) {
   s.payType(t);
}

public void approve() {
   s.approve();
}
public void approvec()
```

```
{
    s.approvec();
    s.M=1;
}

public void reject() {
    s.reject();
}

public void cancel() {
    s.cancel();
}

/*
    g = 1 represents Regular gas
    g = 2 represents Super gas
    g = 3 represents Premium gas
    g = 4 represents Diesel gas
 */
public void selectGas(int g, int M) {
    s.selectGas(g,M);
}

public void startPump() {
    s.startPump();
}

public void pump() {
    s.pump();
}

public void stopPump() {
    s.stopPump();
}

public void receipt() {
    s.receipt();
}

public void noReceipt() {
    s.noReceipt();
}
public void Continue()
{
    s.Continue();
}
```

```
    public void correctPin()
    {
        s.correctPin();
    }
    public void incorrectPin()
    {
        s.incorrectPin();
    }
}
```

State S6

```
package MDA;

/*
    State S6
 */
class S6 extends State {

    S6(MDAEFSM model) {
        super(model);
    }

    /*
        Transition to State S0 and call PrintReceipt() and ReturnCash()
     */
    @Override
    void receipt() {
        if (model.s == model.LS[6]) {
            model.s = model.LS[0];
            model.getOP().PrintReceipt();
            model.getOP().ReturnCash();
        }
    }

    /*
        Transition to State S0 and call ReturnCash()
     */
    @Override
    void noReceipt() {
        if (model.s == model.LS[6]) {
            model.s = model.LS[0];
            model.getOP().ReturnCash();
        }
    }
}
```

State S8

```
package MDA;
```

```
/*
    State S8
 */
class S8 extends State {

    S8(MDAEFSM model) {
        super(model);
    }
    /*
        Transition to State S3 and call DisplayMenu()
     */
    @Override
    void correctPin() {
        if (model.s == model.LS[8]) {
            model.s = model.LS[3];
             System.out.println("Pin number entered correct....Card Approved");
            model.getOP().DisplayMenu();

        }
    }

    /*
        Transition to State S0 and call WrongPinMsg()
     */
    @Override
    void incorrectPin() {
        if (model.s == model.LS[8]) {
            model.s = model.LS[0];
            model.getOP().WrongPinMsg();
        }
    }
}
```

5.6 Strategy Pattern

5.6.1 Class CancelMsg

```
package Strategy;
/*
   Abstract CancelMsg action strategy.

 */
public abstract class CancelMsg
{
   public CancelMsg()
   {
   }
   public abstract void cancelMsg();
}
```

5.6.2 Class CancelMsg1

```
package Strategy;
/*
GasPump1:CancelMsg prints cancellation messsage
 */
public class CancelMsg1 extends CancelMsg
{
   @Override
   public void cancelMsg()
   {
      System.out.println("Cancelling transaction ... ");
   }
}
```

5.6.3 Class CancelMsg2

```
package Strategy;

/*
GasPump2:CancelMsg prints cancellation message
 */
public class CancelMsg2 extends CancelMsg
{
   @Override
   public void cancelMsg()
   {
      System.out.println("Cancelling transaction ... ");
   }
}
```

5.6.4 Class DisplayMenu

```
package Strategy;
import DataRepository.DataStore;
/*
   Abstract DisplayMenu action strategy
```

```
 */
public abstract class DisplayMenu
{
    DataStore data;
    public DisplayMenu(DataStore data)
    {
        this.data = data;
    }
    public abstract void displayMenu();
}
```

5.6.5 Class DisplayMenu1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:DisplayMenu prints the main menu
 */
public class DisplayMenu1 extends DisplayMenu
{
    public DisplayMenu1(DataStore data)
    {
        super(data);
    }
    /*
        Print a menu and display "credit card approved" message
    */
    @Override
    public void displayMenu()
    {
        DataStore1 d = (DataStore1) data;

        System.out.println("Please select gas type:");
        System.out.println(
                "(5) Regular [$" + d.Rprice + "/gal] " +
                        "\n(6) Diesel [$" + d.Sprice + "/gal]");
        System.out.println("Otherwise, select (7) to cancel");
    }
}
```

5.6.6 Class DisplayMenu2

```
package Strategy;

import DataRepository.DataStore;
import DataRepository.DataStore2;

/*
GasPump2:DisplayMenu prints the main menu
```

```
 */
public class DisplayMenu2 extends DisplayMenu
{
   public DisplayMenu2(DataStore data)
   {
      super(data);
   }
   @Override
   public void displayMenu()
   {
      DataStore2 d = (DataStore2) data;
      System.out.println("Please select gas type: ");
      System.out.println(
            "(3) Regular [$" + d.Rprice + "/liter] " +
                  "\n(4) Super [$" + d.Sprice + "/liter] " +
                  "\n(5) Premium [$" + d.Pprice + "/liter]");
      System.out.println("Otherwise, select (6) to cancel");
   }
}
```

5.6.7 Class EnterPinMsg

```
package Strategy;
/*
   Abstract EnterPinMsg action strategy
 */
import DataRepository.DataStore;

public abstract class EnterPinMsg
{
   DataStore data;
   public EnterPinMsg(DataStore data)
   {
      this.data = data;
   }
   public abstract void enterPinMsg();
}
```

5.6.8 Class EnterPinMsg1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;

public class EnterPinMsg1 extends EnterPinMsg
{
   public EnterPinMsg1(DataStore data)
   {
      super(data);
```

```
    }

    @Override
    public void enterPinMsg()
    {
        DataStore1 d = (DataStore1) data;
        System.out.println("Pin entered");
    }
}
```

5.6.9 Class EnterPinMsg2

```
package Strategy;

import DataRepository.DataStore;

public class EnterPinMsg2 extends EnterPinMsg
{
   public EnterPinMsg2(DataStore data)
    {
        super(data);
    }

    @Override
    public void enterPinMsg()
    {

    }
}
```

5.6.10 Class GasPumpedMsg

```
package Strategy;
import DataRepository.DataStore;
/*
    Abstract GasPumpedMsg action strategy
 */
public abstract class GasPumpedMsg
{
    DataStore data;
    public GasPumpedMsg(DataStore data)
    {
        this.data = data;
    }
    public abstract void gasPumpedMsg();
}
```

5.6.11 Class GasPumpedMsg1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
```

```
/*
GasPump1:GasPumpedMsg prints a message that gasoline has been pumped
 */
public class GasPumpedMsg1 extends GasPumpedMsg
{
   public GasPumpedMsg1(DataStore data)
   {
      super(data);
   }
   /*
      Print a message that 1 gallon of gasoline has been pumpped
      along with the total
    */
   @Override
   public void gasPumpedMsg()
   {
      DataStore1 d = (DataStore1) data;
      System.out.println("Pumped 1 gallon of " + d.gasType + " gasoline");
      System.out.println("Total # of gallons pumped: " + d.G);
   }

}
```

5.6.12 Class GasPumpedMsg2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;
/*
GasPump2:GasPumpedMsg prints a message that gasoline has been pumped
 */
public class GasPumpedMsg2 extends GasPumpedMsg
{
   public GasPumpedMsg2(DataStore data)
   {
      super(data);
   }
   /*
      Print a a message that 1 liter of gasoline has been pumped
      along with total number of liters pumped.
    */
   @Override
   public void gasPumpedMsg()
   {
      DataStore2 d = (DataStore2) data;
      System.out.println("Pumped 1 liter of " + d.gasType + " gasoline");
      System.out.println("Total # of liters pumped: " + d.L);
   }
}
```

5.6.13 Class InitializeData

```
package Strategy;
import DataRepository.DataStore;
/*
   Abstract InitializeData action strategy
 */
public abstract class InitializeData
{
   DataStore data;
   public InitializeData(DataStore data)
   {
      this.data = data;
   }
   public abstract void initializeData();
}
```

5.6.14 Class InitializeData1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:InitializeData will initialize the value of price and cash
 */
public class InitializeData1 extends InitializeData
{
   public InitializeData1(DataStore data)
   {
      super(data);
   }
   /*
      Set the price to zero for this transaction
   & cash can't be set since no cash transaction
    */
   @Override
   public void initializeData()
   {
      DataStore1 d = (DataStore1) data;
      d.price = 0;
   }
}
```

5.6.15 Class InitializeData2

```
package Strategy;

import DataRepository.DataStore;
```

```
/*
GasPump2:InitializeData doesn't initilaise pricec & cash
 */
public class InitializeData2 extends InitializeData
{

    public InitializeData2(DataStore data) {
     super(data);
    }
    @Override
    public void initializeData()
    {

    }
}
```

5.6.16 Class PayMsg

```
package Strategy;
/*
    Abstract PayMsg action strategy
 */
public abstract class PayMsg
{
    public PayMsg()
    {
    }
    public abstract void payMsg();
}
```

5.6.17 Class PayMsg1

```
package Strategy;
/*
GasPump1:PayMsg prompts for payment selection
 */
public class PayMsg1 extends PayMsg
{
    @Override
    public void payMsg()
    {
        System.out.println("Please select payment type");

    }
}
```

5.6.18 Class PayMsg2

```
package Strategy;

/*
```

```
GasPump2:PayMsg prompts for payment selection
 */
public class PayMsg2 extends PayMsg {

    @Override
    public void payMsg() {
        System.out.println("Please select payment type: ");
    }
}
```

5.6.19 Class PrintReceipt

```
package Strategy;
import DataRepository.DataStore;

 // Abstract for PrintReceipt

public abstract class PrintReceipt
{
    DataStore data;
    public PrintReceipt(DataStore data)
    {
        this.data = data;
    }
    public abstract void printReceipt();
}
```

5.6.20 Class PrintReceipt1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
    // generating and printing receipt for GasPump1

public class PrintReceipt1 extends PrintReceipt
{
    public PrintReceipt1(DataStore data)
    {
        super(data);
    }
    @Override
    public void printReceipt()
    {
        System.out.println("\nFollowing is the Receipt");
        DataStore1 d = (DataStore1) data;
        System.out.println(d.G + " gallons of " + d.gasType + " gas @ $" + d.price + "/gallon");
        System.out.println("Total: $" + d.total);
    }
}
```

5.6.21 Class PrintReceipt2

```
package Strategy;

import DataRepository.DataStore;
import DataRepository.DataStore2;


// generating and printing receipt for GasPump2

public class PrintReceipt2 extends PrintReceipt
{
   public PrintReceipt2(DataStore data)
   {
      super(data);
   }
   @Override
   public void printReceipt()
   {
      System.out.println("\nFollowing is the Receipt");
      DataStore2 d = (DataStore2) data;
      System.out.println(d.L + " liters of " + d.gasType + " gas @ $" + d.price + "/liter");
      System.out.println("Cash inserted: $" + d.cash);
      System.out.println("Total: $" + (float) d.total);
   }
}
```

5.6.22 Class PumpGasUnit

```
package Strategy;
import DataRepository.DataStore;
/*
   Abstract PumpGasUnit action strategy
 */
public abstract class PumpGasUnit
{
   DataStore data;
   public PumpGasUnit(DataStore data)
   {
      this.data = data;
   }
   public abstract void pumpGasUnit();
}
```

5.6.23 Class PumpGasUnit1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:PumpGasUnit for pumping a gallon of gas
 */
```

```
public class PumpGasUnit1 extends PumpGasUnit
{
    public PumpGasUnit1(DataStore data)
    {
        super(data);
    }
    /*
        Read and update attributes corresponding to pumping a gallon of gas
     */
    @Override
    public void pumpGasUnit()
    {
        DataStore1 d = (DataStore1) data;
        d.G++;
        d.total = d.price * d.G;
    }
}
```

5.6.24 Class PumpGasUnit2

```
package Strategy;

import DataRepository.DataStore;
import DataRepository.DataStore2;

/*
    GasPump2 PumpGasUnit action for pumping a liter of gas
 */
public class PumpGasUnit2 extends PumpGasUnit {

    public PumpGasUnit2(DataStore data) {
        super(data);
    }
    /*
    Read and update attributes corresponding to pumping a liter of gas
 */

    @Override
    public void pumpGasUnit() {
        DataStore2 d = (DataStore2) data;

        d.L++;
        d.total = d.price * d.L;
    }
}
```

5.6.25 Class ReadyMsg

```
package Strategy;
import DataRepository.DataStore;
```

```
/*
   Abstract ReadyMsg action strategy
 */
public abstract class ReadyMsg
{
   DataStore data;
   public ReadyMsg(DataStore data)
   {
      this.data = data;
   }
   public abstract void readyMsg();
}
```

5.6.26 Class ReadyMsg1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:ReadyMsg prints a ready message
 */
public class ReadyMsg1 extends ReadyMsg
{
   public ReadyMsg1(DataStore data)
   {
      super(data);
   }
   /*
      display message GasPump is ready to dispense 1 gallon of selected gasline
    */
   @Override
   public void readyMsg()
   {
      System.out.println("\nReady to Dispense Gas");
      DataStore1 d = (DataStore1) data;
      System.out.println("Select (9) to dispense 1 gallon of " + d.gasType + " gasoline");
      System.out.println("Otherwise, select (s) to stop");
   }
}
```

5.6.27 Class ReadyMsg2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;
/*
GasPump2:ReadyMsg print a ready message
 */
public class ReadyMsg2 extends ReadyMsg
{
    public ReadyMsg2(DataStore data)
    {
        super(data);
    }
    /*
    display message GasPump is ready to dispense 1 liter of selected gasline
 */
    @Override
    public void readyMsg()
    {
        System.out.println("\nREADY TO DISPENSE FUEL");
        DataStore2 d = (DataStore2) data;
        System.out.println("Select (8) to dispense 1 liter of " + d.gasType + " gasoline");
        System.out.println("Otherwise, select (9) to stop");
    }
}
```

5.6.28 Class RejectMsg

```
package Strategy;
/*
    Abstract RejectMsg action strategy
 */
public abstract class RejectMsg
{
    public RejectMsg()
    {
    }
    public abstract void rejectMsg();
}
```

5.6.29 Class RejectMsg1

```
package Strategy;
/*
GasPump1:RejectMsg prints credit card rejection message
 */
public class RejectMsg1 extends RejectMsg
{
    /*
        display credit card rejected message
     */
    @Override
    public void rejectMsg()
    {
        System.out.println("CREDIT CARD REJECTED");
        System.out.println("Cancelling transaction ...");
    }
}
```

5.6.30 Class RejectMsg2

```
package Strategy;
/*
GasPump2:RejectMsg prints credit card rejection message
 */
public class RejectMsg2 extends RejectMsg
{
    /*
        display credit card rejected message
    */
    @Override
    public void rejectMsg()
    {
        System.out.println("CREDIT CARD REJECTED");
        System.out.println("Cancelling transaction ...");
    }
}
```

5.6.31 Class ReturnCash

```
package Strategy;
import DataRepository.DataStore;
/*
    Abstract ReturnCash action strategy
 */
public abstract class ReturnCash
{
    DataStore data;
    /*public ReturnCash()
    {
    }*/
    public ReturnCash(DataStore data)
    {
        this.data = data;
    }
    public abstract void returnCash();
}
```

5.6.32 Class ReturnCash1

```
package Strategy;

import DataRepository.DataStore;

/*
GasPump1:ReturnCash does not support since cash not used
 */
public class ReturnCash1 extends ReturnCash
{
    /*
        GasPump1 does not support payment with cash so no action
    */
    public ReturnCash1(DataStore data)
    {
        super(data);
    }
    @Override
    public void returnCash()
    {

    }
}
```

5.6.33 Class ReturnCash2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;
/*
GasPump2:ReturnCash returns the outstanding amount of cash
 */
public class ReturnCash2 extends ReturnCash {

    public ReturnCash2(DataStore data)
    {
        super(data);
    }
    /*
        This method first reads the shared data structure to calculate the amount of cash to return
        If there is any amount greater than $0 that is owed to the user, print a message indicating so,
        and return the cash
        Then, reset the data structure "cash" attribute to 0
     */
    @Override
    public void returnCash()
    {
        DataStore2 d = (DataStore2) data;
        float cash_return = d.cash - d.total;
        if (cash_return > 0)
        {
            System.out.println("Cash to be returned: $" + cash_return);
            System.out.println("Returning $" + cash_return);
        }
        else
        {
            System.out.println("No cash to return");
        }
        d.cash = 0;
        System.out.println("Transaction completed");
    }
}
```

5.6.34 Class SetInitialValues

```
package Strategy;
import DataRepository.DataStore;
/*
    Abstract SetInitialValues action strategy
 */
public abstract class SetInitialValues
{
    DataStore data;
    public SetInitialValues(DataStore data)
    {
        this.data = data;
    }
    public abstract void setInitialValues();
}
```

5.6.35 Class SetInitialValues1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:SetInitialValues will initialize the value of G and total
 */
public class SetInitialValues1 extends SetInitialValues
{
    public SetInitialValues1(DataStore data)
    {
        super(data);
    }
    /*
        Set the number of gallons pumped and payment balance initially to zero for this transaction
     */
    @Override
    public void setInitialValues()
    {
        DataStore1 d = (DataStore1) data;
        d.G = 0;
        d.total = 0;
    }
}
```

5.6.36 Class SetInitialValues2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;
/*
```

```
GasPump2:SetInitialValues will initialize the values of L and total
 */
public class SetInitialValues2 extends SetInitialValues
{
   public SetInitialValues2(DataStore data)
   {
      super(data);
   }
   /*
      Set the number of liters pumped and payment balance initially to zero for this transaction
    */
   @Override
   public void setInitialValues()
   {
      DataStore2 d = (DataStore2) data;
      d.L = 0;
      //d.total = 0;
   }
}
```

5.6.37 Class SetPrice

```
package Strategy;
import DataRepository.DataStore;
/*
   Abstract SetPrice action strategy
 */
public abstract class SetPrice
{
   DataStore data;
   public SetPrice(DataStore data)
   {
      this.data = data;
   }
   public abstract void setPrice(int g, int M);
}
```

5.6.38 Class SetPrice1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*
GasPump1:SetPrice updates the values of Rprice and Sprice based on a and b respectively
 */
public class SetPrice1 extends SetPrice
{

   public SetPrice1(DataStore data)
   {
        super(data);
    }
/*
        Set the price per gallon for this transaction according to the type of gas which was selected
to be pumped
        g = 1: Regular gas
        g = 4: Diesel gas
         */
    @Override
    public void setPrice(int g, int M) {
        DataStore1 d = (DataStore1) data;
        if (g == 1)
        {
            // Regular selected
            d.price = d.Rprice;
            d.gasType = "Regular";
            M=1;
        }
        else if (g == 4)
        {
            // Diesel selected
            d.price = d.Sprice;
            d.gasType = "Diesel";
            M=1;
        }
        System.out.println(d.gasType + " gasoline selected @ price of $" + d.price + "/gallon");
        System.out.println("Select (8) to start the pump");
    }
}
```

5.6.39 Class SetPrice2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;

/*
 * GasPump2: SetPrice updates the values of Rprice, Sprice and Pprice based on a, b and c
respectively
 */
public class SetPrice2 extends SetPrice
{

    public SetPrice2(DataStore data)
    {
        super(data);
    }
    /*
```

```
      Set the price per gallon for this transaction according to the type of gas which was selected to be pumped
      g = 1: Regular gas
      g = 2: Super gas
      g = 3: Premium gas
   */
   @Override
   public void setPrice(int g, int M)
   {
      DataStore2 d = (DataStore2) data;
      if (g == 1 )
      {
         // Regular selected
         d.price = d.Rprice;
         d.gasType = "Regular";
      }
      else if (g == 2)
      {
         // Super selected
         d.price = d.Sprice;
         d.gasType = "Super";
      }
      else if (g == 3)
      {
         // Premium selected
         d.price = d.Pprice;
         d.gasType = "Premium";
      }
      System.out.println(d.gasType + " gasoline selected @ price of $" + d.price + "/liter");
      System.out.println("Select (7) to start the pump");
   }
}
```

5.6.40 Class StopMsg

```
package Strategy;
/*
   Abstract StopMsg action strategy

 */
public abstract class StopMsg
{
   public StopMsg()
   {
   }
   public abstract void stopMsg();
}
```

5.6.41 Class StopMsg1

```
package Strategy;

/*
GasPump1:StopMsg displays a message stating that the pump1 is stopping.
 */
public class StopMsg1 extends StopMsg
{
    @Override
    public void stopMsg()
    {
        System.out.println("Stopping the pump ...");
    }
}
```

5.6.42 Class StopMsg2

```
package Strategy;

/*
GasPump2:StopMsg displays a message stating that the pump2 is stopping
 */
public class StopMsg2 extends StopMsg {

    @Override
    public void stopMsg() {
        System.out.println("Stopping the Pump ...");
    }
}
```

5.6.43 Class StoreCash

```
package Strategy;
import DataRepository.DataStore;
/*
    Abstract StoreCash action strategy
 */
public abstract class StoreCash
{
    DataStore data;
    /*public StoreCash()
    {
    }*/
    public StoreCash(DataStore data)
    {
        this.data = data;
    }
    public abstract void storeCash();
}
```

5.6.44 Class StoreCash1

```
package Strategy;
```

```
import DataRepository.DataStore;

/*
GasPump1:StoreCash not supported
 */
public class StoreCash1 extends StoreCash
{
    /*
        GasPump1 does not bolster payment with cash, and so this method should never be invoked
        by GasPump1
    */
    public StoreCash1(DataStore data)
    {
        super(data);
    }
    @Override
    public void storeCash() {

    }
}
```

5.6.45 Class StoreCash2

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore2;
/*
GasPump2:StoreCash updates the shared data structure with the inserted cash amount
    during each transaction
 */
public class StoreCash2 extends StoreCash
{
    public StoreCash2(DataStore data)
    {
        super(data);
    }
    /*
        Read the temporarily cash information and store it in the cash attribute of the shared data
structure.
        print the amount of cash that was inserted
     */
    @Override
    public void storeCash()
    {
        DataStore2 d = (DataStore2) data;
        d.cash = d.temp_cash;
        System.out.println("Inserted/Deposited Sum of cash: $" + d.cash);
    }
}
```

5.6.46 Class StorePin

```
package Strategy;
import DataRepository.DataStore;
/*
    Abstract StorePin action strategy
 */
public abstract class StorePin
{
    DataStore data;
    public StorePin(DataStore data)
    {
        this.data = data;
    }
    public abstract void storePin();
}
```

5.6.47 Class StorePin1

```
package Strategy;
import DataRepository.DataStore;
import DataRepository.DataStore1;
/*

 */
public class StorePin1 extends StorePin
{
    public StorePin1(DataStore data)
    {
        super(data);
    }

    @Override
    public void storePin()
    {
        DataStore1 d = (DataStore1) data;
        d.pin = d.temp_p;
        System.out.println("Pin stored: $" + d.pin);
    }
}
```

5.6.48 Class StorePin2

```
package Strategy;

import DataRepository.DataStore;

public class StorePin2 extends StorePin
{
    public StorePin2(DataStore data)
    {
        super(data);
    }
    @Override
    public void storePin() {

    }
}
```

5.6.49 Class WrongPinMsg

```
package Strategy;

import DataRepository.DataStore;

/*
    Abstract CancelMsg action strategy.

 */
public abstract class WrongPinMsg
{
    DataStore data;
    public WrongPinMsg()
    {

    }
    public abstract void wrongPinMsg();

}
```

5.6.50 Class WrongPinMsg1

```
package Strategy;

import DataRepository.DataStore;

public class WrongPinMsg1 extends WrongPinMsg
{

    @Override
    public void wrongPinMsg() {
        System.out.println("Pin Number entered is wrong!!!");
    }
}
```

5.6.51 Class WrongPinMsg2

```
package Strategy;

import DataRepository.DataStore;
```

```
/*
GasPump2:WrongPinMsg does not support since debit card is not used
 */
public class WrongPinMsg2 extends WrongPinMsg
{
    /*
        GasPump2 does not support payment with debit card so no action
    */

    @Override
    public void wrongPinMsg()
    {

    }
}
```

5.6.52 Class StorePrices

```
package Strategy;

import DataRepository.DataStore;

/*
    Abstract StorePrices action strategy
 */
public abstract class StorePrices {
    DataStore data;

    public StorePrices(DataStore data) {
        this.data = data;
    }

    public abstract void storePrices();
}
```

5.6.53 Class StorePrices1

```
package Strategy;

import DataRepository.DataStore;
import DataRepository.DataStore1;

/*
    GasPump1 StorePrices action stores the "a" and "b" price parameters specified by
    method "Activate" of the InputProcessor for GasPump1
 */
public class StorePrices1 extends StorePrices
{
    public StorePrices1(DataStore data)
    {
        super(data);
    }
    /*
        Read the temporary variables "a" and "b"
        & Initialize the gas prices
     */
    @Override
    public void storePrices()
    {
        DataStore1 d = (DataStore1) data;
        d.Rprice = d.a;
        d.Sprice = d.b;
        System.out.println("GasPump1 triggered successfully!!!");
    }
}
```

5.6.54 Class StorePrices2

```
package Strategy;

import DataRepository.DataStore;
import DataRepository.DataStore2;

/*
    GasPump2 StorePrices action responsible for storing the "a", "b", "c"  price parameters
specified by
    method "Activate" of the InputProcessor for GasPump2
 */
public class StorePrices2 extends StorePrices
{

    public StorePrices2(DataStore data)
    {
        super(data);
    }
    /*
        Read the temporary variables "a", "b", and "c"
        & initialize the gas prices
     */
    @Override
    public void storePrices()
    {
        DataStore2 d = (DataStore2) data;
        d.Rprice = d.a;
        d.Sprice = d.b;
        d.Pprice = d.c;
        System.out.println("GasPump2 triggered successfully!!!");
    }
}
```

5.7 Abstract Factory Pattern

Abstract Factory

```
package AbstractFactory;

import DataRepository.DataStore;
import Strategy.CancelMsg;
import Strategy.DisplayMenu;
import Strategy.EnterPinMsg;
import Strategy.GasPumpedMsg;
import Strategy.InitializeData;
import Strategy.PayMsg;
import Strategy.PrintReceipt;
import Strategy.PumpGasUnit;
import Strategy.ReadyMsg;
import Strategy.RejectMsg;
import Strategy.ReturnCash;
import Strategy.SetInitialValues;
import Strategy.SetPrice;
import Strategy.StopMsg;
import Strategy.StoreCash;
import Strategy.StorePin;
import Strategy.StorePrices;
import Strategy.WrongPinMsg;

/*

        takes all the ConcreteFactory classes in 1 AbstractFactory class
        Consists of the methods that concreteFactory class needs to implement

 */
public abstract class AbstractFactory {

    public abstract DataStore getDataObj();

    public abstract CancelMsg getCancelMsg();

    public abstract DisplayMenu getDisplayMenu();

    public abstract GasPumpedMsg getGasPumpedMsg();

    public abstract PayMsg getPayMsg();

    public abstract PrintReceipt getPrintReceipt();
    public abstract PumpGasUnit getPumpGasUnit();

    public abstract ReadyMsg getReadyMsg();

    public abstract RejectMsg getRejectMsg();

    public abstract ReturnCash getReturnCash();

    public abstract SetInitialValues getSetInitialValues();

    public abstract SetPrice getSetPrice();

    public abstract StopMsg getStopMsg();

    public abstract StoreCash getStoreCash();

    public abstract StorePrices getStorePrices();

    public abstract WrongPinMsg getWrongPinMsg();

    public abstract EnterPinMsg getEnterPinMsg();

    public abstract InitializeData getInitializeData();

    public abstract StorePin getStorePin();


}
```

Concrete Factory 1

```
package AbstractFactory;

import DataRepository.DataStore;
import DataRepository.DataStore1;
import Strategy.CancelMsg;
import Strategy.CancelMsg1;
import Strategy.DisplayMenu;
import Strategy.DisplayMenu1;
import Strategy.EnterPinMsg;
import Strategy.EnterPinMsg1;
import Strategy.GasPumpedMsg;
import Strategy.GasPumpedMsg1;
import Strategy.InitializeData;
import Strategy.InitializeData1;
import Strategy.PayMsg;
import Strategy.PayMsg1;
import Strategy.PrintReceipt;
import Strategy.PrintReceipt1;
import Strategy.PumpGasUnit;
import Strategy.PumpGasUnit1;
import Strategy.ReadyMsg;
import Strategy.ReadyMsg1;
import Strategy.RejectMsg;
import Strategy.RejectMsg1;
import Strategy.ReturnCash;
import Strategy.ReturnCash1;
import Strategy.SetInitialValues;
import Strategy.SetInitialValues1;
import Strategy.SetPrice;
import Strategy.SetPrice1;
import Strategy.StopMsg;
import Strategy.StopMsg1;
import Strategy.StoreCash;
import Strategy.StoreCash1;
import Strategy.StorePin;
import Strategy.StorePin1;
import Strategy.StorePrices;
import Strategy.StorePrices1;
import Strategy.WrongPinMsg;
import Strategy.WrongPinMsg1;
/*
    This class is the factory that produces the necessary driver objects for GasPump1
    Instantiates the proper action strategies with the shared data structure
    OutputProcessor object will be instantiated with an object of this class when it needs to
    display output for GasPump1. Output processor will call the methods provided by this class in
order to bind
    GasPump1 specific actions.
 */
public class ConcreteFactory1 extends AbstractFactory {
    private DataStore data;

    public ConcreteFactory1() {
    // create DataStore object
        this.data  = new DataStore1();
    }
    // Returns the shared data structure
    @Override
    public DataStore getDataObj() {
        return this.data;
    }
    /*
        Returns the CancelMsg class
     */
    @Override
    public CancelMsg getCancelMsg() {
        return new CancelMsg1();
    }
    /*
        Returns the DisplayMenu class
     */
    @Override
    public DisplayMenu getDisplayMenu() {
        return new DisplayMenu1(this.data);
    }
    /*
        Returns the GasPumpedMsg class
     */
    @Override
    public GasPumpedMsg getGasPumpedMsg() {
        return new GasPumpedMsg1(this.data);
    }

    /*
        Returns message to ask the user for Payment
     */
    @Override
    public PayMsg getPayMsg() {
        return new PayMsg1();
    }

    /*
        Returns the PrintReceipt class which is responsible to print receipt
     */
    @Override
    public PrintReceipt getPrintReceipt() {
        return new PrintReceipt1(this.data);
    }

    /*
        Returns the PumpGasUnit class which is used to "pumping" a unit of gas
     */
    @Override
    public PumpGasUnit getPumpGasUnit() {
        return new PumpGasUnit1(this.data);
    }

    /*
        Returns the ReadyMsg class which prompts the user to start pumping gas
     */
    @Override
    public ReadyMsg getReadyMsg() {
        return new ReadyMsg1(this.data);
    }

    /*
        Returns the RejectMsg class and displays card rejected message
     */
    @Override
    public RejectMsg getRejectMsg() {
        return new RejectMsg1();
    }

    /*
        Returns the ReturnCash but cash is not a supported payment
     */
    @Override
    public ReturnCash getReturnCash() {
        return new ReturnCash1(this.data);
    }

    /*
        Returns the SetInitialValues class for initializing
     */
    @Override
    public SetInitialValues getSetInitialValues() {
        return new SetInitialValues1(this.data);
    }

    /*
        Returns the SetPrice class to set the price of Gas
     */
    @Override
    public SetPrice getSetPrice() {
        return new SetPrice1(this.data);
    }

    /*
        Returns the StopMsg class which informs the user that gaspump stopped
     */
    @Override
    public StopMsg getStopMsg() {
     return new StopMsg1();
    }
    /*
```

```
        Returns the StoreCash action but as GasPump1 does not support cash as payment so will
have an empty body for GasPump1
     */
    @Override
    public StoreCash getStoreCash() {
        return new StoreCash1(this.data);
    }
    /*
        Returns the StorePrices action for storing data
     */
    @Override
    public StorePrices getStorePrices() {
        return new StorePrices1(this.data);
    }
    @Override
    public StorePin getStorePin() {
        return new StorePin1(this.data);
    }
    @Override
    public EnterPinMsg getEnterPinMsg() {
        return new EnterPinMsg1(this.data);
    }
    @Override
    public WrongPinMsg getWrongPinMsg() {
       return new WrongPinMsg1();
    }
    @Override
    public InitializeData getInitializeData() {
       return new InitializeData1(this.data);
    }
}
```

Concrete Factory 2

```
package AbstractFactory;

import DataRepository.DataStore;
import DataRepository.DataStore2;
import Strategy.CancelMsg;
import Strategy.CancelMsg2;
import Strategy.DisplayMenu;
import Strategy.DisplayMenu2;
import Strategy.EnterPinMsg;
import Strategy.EnterPinMsg2;
import Strategy.GasPumpedMsg;
import Strategy.GasPumpedMsg2;
import Strategy.InitializeData;
import Strategy.InitializeData2;
import Strategy.PayMsg;
import Strategy.PayMsg2;
import Strategy.PrintReceipt;
import Strategy.PrintReceipt2;
import Strategy.PumpGasUnit;
import Strategy.PumpGasUnit2;
import Strategy.ReadyMsg;
import Strategy.ReadyMsg2;
import Strategy.RejectMsg;
import Strategy.RejectMsg2;
import Strategy.ReturnCash;
import Strategy.ReturnCash2;
import Strategy.SetInitialValues;
import Strategy.SetInitialValues2;
import Strategy.SetPrice;
import Strategy.SetPrice2;
import Strategy.StopMsg;
import Strategy.StopMsg2;
import Strategy.StoreCash;
import Strategy.StoreCash2;
import Strategy.StorePin;
import Strategy.StorePin2;
import Strategy.StorePrices;
import Strategy.StorePrices2;
import Strategy.WrongPinMsg;
import Strategy.WrongPinMsg2;
/*
    This class is the factory that produces the necessary driver objects for GasPump2
    Output processor will call the methods provided by this class in order to bind
    GasPump2 specific actions.
*/
public class ConcreteFactory2 extends AbstractFactory {
    private DataStore data;
    public ConcreteFactory2() {
        // Create DataStore object
        data  = new DataStore2();
    }
    /*
        Returns the shared data structure
     */
    @Override
    public DataStore getDataObj() {
        return this.data;
    }

    /*
```

```
        Returns the CancelMsg class to display cancel message
     */
    @Override
    public CancelMsg getCancelMsg() {
        return new CancelMsg2();
    }

    /*
        Returns the DisplayMenu class to display menu for GasPump2
     */
    @Override
    public DisplayMenu getDisplayMenu() {
        return new DisplayMenu2(this.data);
    }

    /*
        Returns the GasPumpedMsg class that displays the unit of gas has been pumped
     */
    @Override
    public GasPumpedMsg getGasPumpedMsg() {
        return new GasPumpedMsg2(this.data);
    }

    /*
        Returns the Payment prompt message
     */
    @Override
    public PayMsg getPayMsg() {
        return new PayMsg2();
    }

    /*
        Returns the PrintReceipt class for printing receipt
     */
    @Override
    public PrintReceipt getPrintReceipt() {
        return new PrintReceipt2(this.data);
    }

    /*
        Returns the PumpGasUnit class which is responsible to "pumping" a unit of gas for
GasPump2
     */
    @Override
    public PumpGasUnit getPumpGasUnit() {
        return new PumpGasUnit2(this.data);
    }

    /*
        Returns the ReadyMsg class which tells user to start pumping
     */
    @Override
    public ReadyMsg getReadyMsg() {
        return new ReadyMsg2(this.data);
    }

    /*
        Returns the RejectMsg class for GasPump2 does not support credit card as payment so
action strategy method will have an empty body
     */
    @Override
    public RejectMsg getRejectMsg() {
        return new RejectMsg2();
    }

    /*
        Returns the ReturnCash action will display return cash along with any cash which is to be
returned
     */
    @Override
    public ReturnCash getReturnCash() {
        return new ReturnCash2(this.data);
    }

    /*
        Returns the SetInitialValues class
    */
    @Override
    public SetInitialValues getSetInitialValues() {
        return new SetInitialValues2(this.data);
    }

    /*
        Returns the SetPrice class for setting the price of gas
    */
    @Override
    public SetPrice getSetPrice() {
        return new SetPrice2(this.data);
    }

    /*
        Returns the StopMsg class which displays GasPump has stopped
     */
    @Override
    public StopMsg getStopMsg() {
        return new StopMsg2();
    }

    /*
        Returns the StoreCash class which provides the appropriate action for storing cash
        on GasPump2.
     */
    @Override
    public StoreCash getStoreCash() {
        return new StoreCash2(this.data);
    }

    /*
        Returns the StorePrices class which stores temporary data
     */
    @Override
    public StorePrices getStorePrices() {
        return new StorePrices2(this.data);
    }

    @Override
    public StorePin getStorePin() {
        return new StorePin2(this.data);
    }
    @Override
    public EnterPinMsg getEnterPinMsg() {
        return new EnterPinMsg2(this.data);
    }
    @Override
    public WrongPinMsg getWrongPinMsg() {
       return new WrongPinMsg2();
    }
    @Override
    public InitializeData getInitializeData() {
       return new InitializeData2(this.data);
    }

}
```

Conclusion:

In this project I have implemented two Gas Pump Components using three patterns State Pattern, Strategy pattern and Abstract Factory Pattern. I have documented two given scenarios using separate sequence diagrams for each action. The coding of the System is done using Object Oriented programming language (JAVA). The source files of the system are also included in the project.