Chethan Bhaskar

MDA-EFSM GasPump Project

CS-586 Software System Architecture

Spring 2017

Mail id: cbhaskar@hawk.iit.edu

MDA-EFSM model for the *GasPump* Components

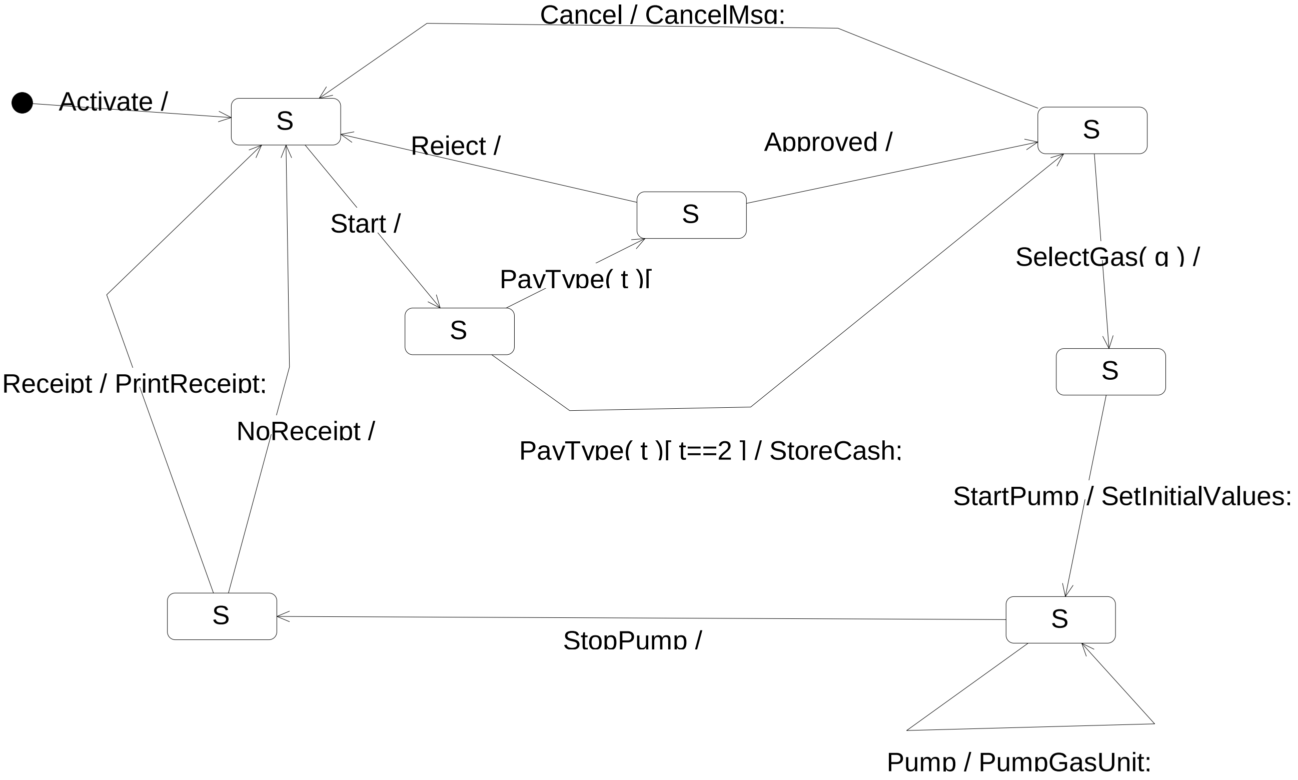
MDA-EFSM meta events

Activate()	
Start()	
PayType(int t)	//credit: t=1; cash: t=2
Reject()	
Cancel()	
Approved()	
StartPump()	
Pump()	
StopPump()	
SelectGas(int g)	
Receipt()	
NoReceipt()	

MDA-EFSM meta actions

StoreData	// 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)	// set the price for the gas identified by g identifier
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

State Diagram



Pseudo-code

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();
}

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

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

Super() {
        m->SelectGas(2)
}

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

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

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

Note:
m: is a pointer to the MDA-EFSM object
d: is a pointer to the Data Store object

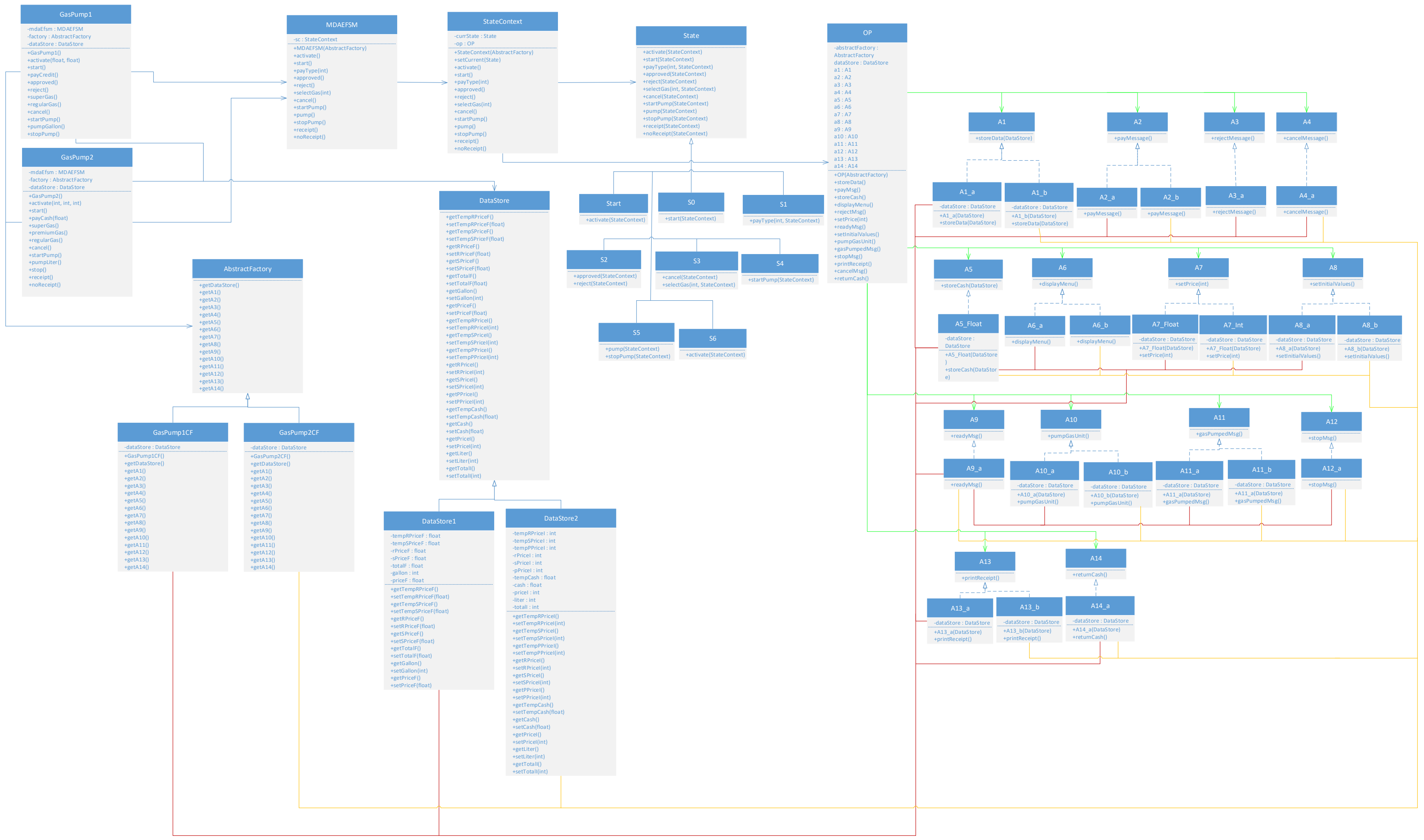
Operations of the Input Processor (GasPump-2)

```
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()
        }
}

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

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

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

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

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

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

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

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

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

NoReceipt() {
        m->NoReceipt();
}
```

Note:
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

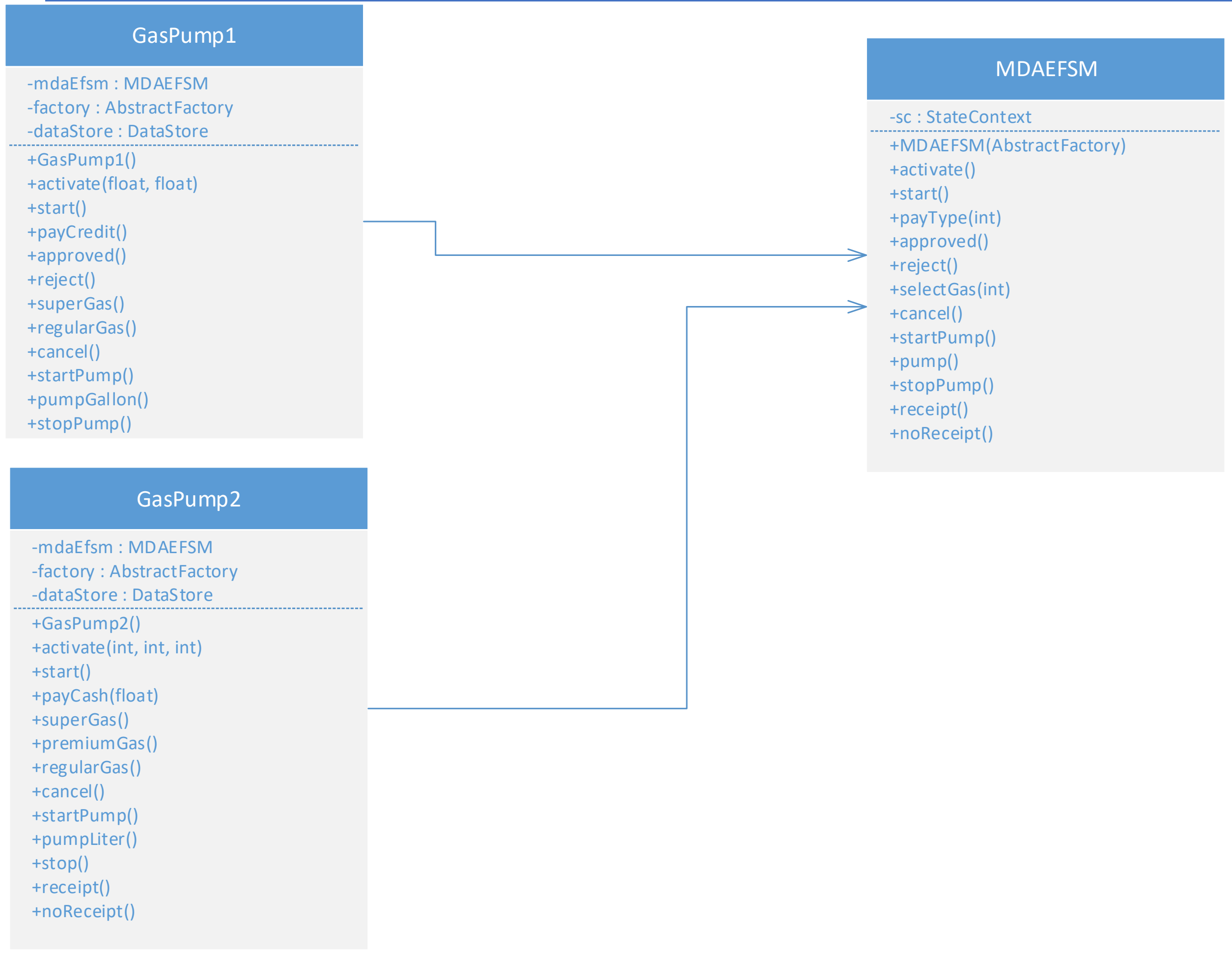
Class Diagram



Note: Lines are colored to represent and distinguish relations between the classes .

Classes, Responsibilities and Design Patterns implemented

1. *GasPump and MDAEFSM:*



❖ GasPump1

Purpose	This class represents GasPump1 component Implementation which addresses all the needs of GasPump1 operations.
Member variables	mdaefsm is a pointer to MDAEFSM class. dataStore is a pointer to DataStore class. abstractFactory is a pointer to AbstractFactory class.
Member functions	
GasPump1()	To initialize abstract factory, datastore and mdaefsm objects
activate(float,float)	To activate the gas pump with initial parameters used to set gas cost
start()	To start the gas pump component
payCredit()	To make the payment for gas through credit card.
cancel()	To cancel the transaction
approved()	To approve the credit card.
reject()	To reject the credit card.
superGas()	To select Super gas type

regularGas()	To select Regular gas type
startPump()	To start pumping gas
pumpGallon()	To dispose a gallon of gas
stopPump()	To stop pump gas

❖ GasPump2

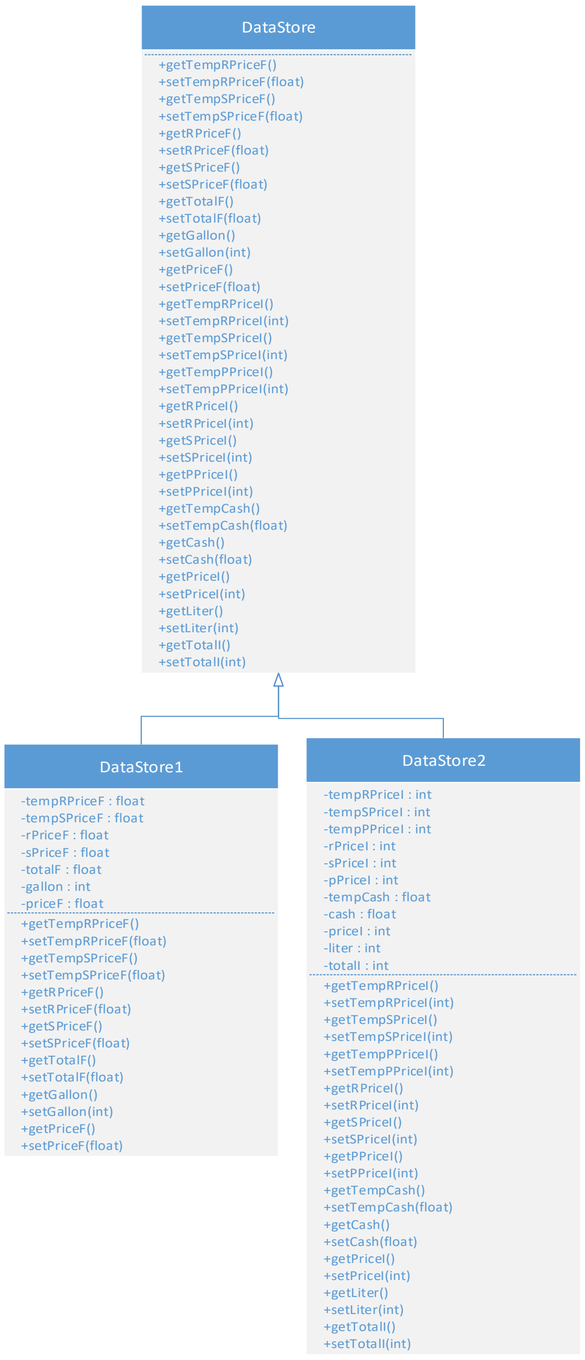
Purpose	This class represents GasPump2 component Implementation which addresses all the needs of GasPump2 operations.
Member variables	mdaefsm is a pointer to MDAEFSM class. dataStore is a pointer to DataStore class. abstractFactory is a pointer to AbstractFactory class.
Member functions	
GasPump2()	To initialize abstract factory, datastore and mdaefsm objects
activate(int,int,int)	To activate the gas pump with initial parameters used to set gas cost
start()	To start the gas pump component
payCash(int)	To make the payment for gas through cash where parameter represents cash entered.
cancel()	To cancel the transaction
premiumGas()	To select Premium gas type
superGas()	To select Super gas type
regularGas()	To select Regular gas type
startPump()	To start pumping gas
pumpLiter()	To dispose a liter of gas
stop()	To stop pump gas
receipt()	To request a receipt
noReceipt()	To not provide a receipt

❖ MDAEFSM

Purpose	This class represents platform independent logic for all its clients i.e.,GasPump1, GasPump2. Seperation of this functionality allows for reduced effort during maintenance and when new clients are added.
Member variables	sc is a pointer to StateContext class.
Member functions	
MDAEFSM()	To initialize sc object by passing factory object
activate()	Call activate() through sc of StateContext class
start()	Call start() through sc of StateContext class
payType()	Call payType() through sc of StateContext class
approved()	Call approved() through sc of StateContext class
reject()	Call reject() through sc of StateContext class
cancel()	Call cancel () through sc of StateContext class
selectGas(int)	Call selectGas () through sc of StateContext class
startPump()	Call startPump () through sc of StateContext class

pump ()	Call pump () through sc of StateContext class
stopPump()	Call stopPump () through sc of StateContext class
receipt()	Call receipt () through sc of StateContext class
noReceipt()	Call noReceipt () through sc of StateContext class

2. DataStore



❖ DataStore

Purpose	This class represents the abstract class for the Data Store which is used to group all the various data store concrete classes within the implementation.
Member variables	None
Member functions	
getTempRPriceF()	Return default value 0, child class has the implementation
setTempRPriceF(float p)	Abstract method, child class has the implementation
getTempSPriceF()	Return default value 0, child class has the implementation
setTempPriceF(float p)	Abstract method, child class has the implementation
getRPriceF()	Return default value 0, child class has the implementation
setRPriceF(float p)	Abstract method, child class has the implementation
getSpriceF()	Return default value 0, child class has the implementation
setSPriceF(float p)	Abstract method, child class has the implementation
getTotalF()	Return default value 0, child class has the implementation
setTotalF(float t)	Abstract method, child class has the implementation
getGallon()	Return default value 0, child class has the implementation
setGallon(int g)	Abstract method, child class has the implementation
getPriceF()	Return default value 0, child class has the implementation
setPriceF(float p)	Abstract method, child class has the implementation

❖ DataStore1

Purpose	This class represents the concrete implementation of Data Store which handles GasPump1 specific storage operations.
Member variables	tempRPriceF : temporary float variable to store Regular gas type price tempSPriceF : temporary float variable to store Super gas type price rPriceF : permanent float variable to store Regular gas price sPriceF : permanent float variable to store Super gas type price totalF : float variable to store total value after gas has been pumped gallon : integer variable to store gallons pumped priceF : float variable to store the price depending on the gas type chosen.
Member functions	
getTempRPriceF()	Getter method to return tempRPriceF value
setTempRPriceF(float p)	Setter method to store tempRpriceF value
getTempSPriceF()	Getter method to return tempSPriceF value
setTempPriceF(float p)	Setter method to store tempSpriceF value
getRPriceF()	Getter method to return rPriceF value
setRPriceF(float p)	Setter method to store rpriceF value
getSpriceF()	Getter method to return sPriceF value

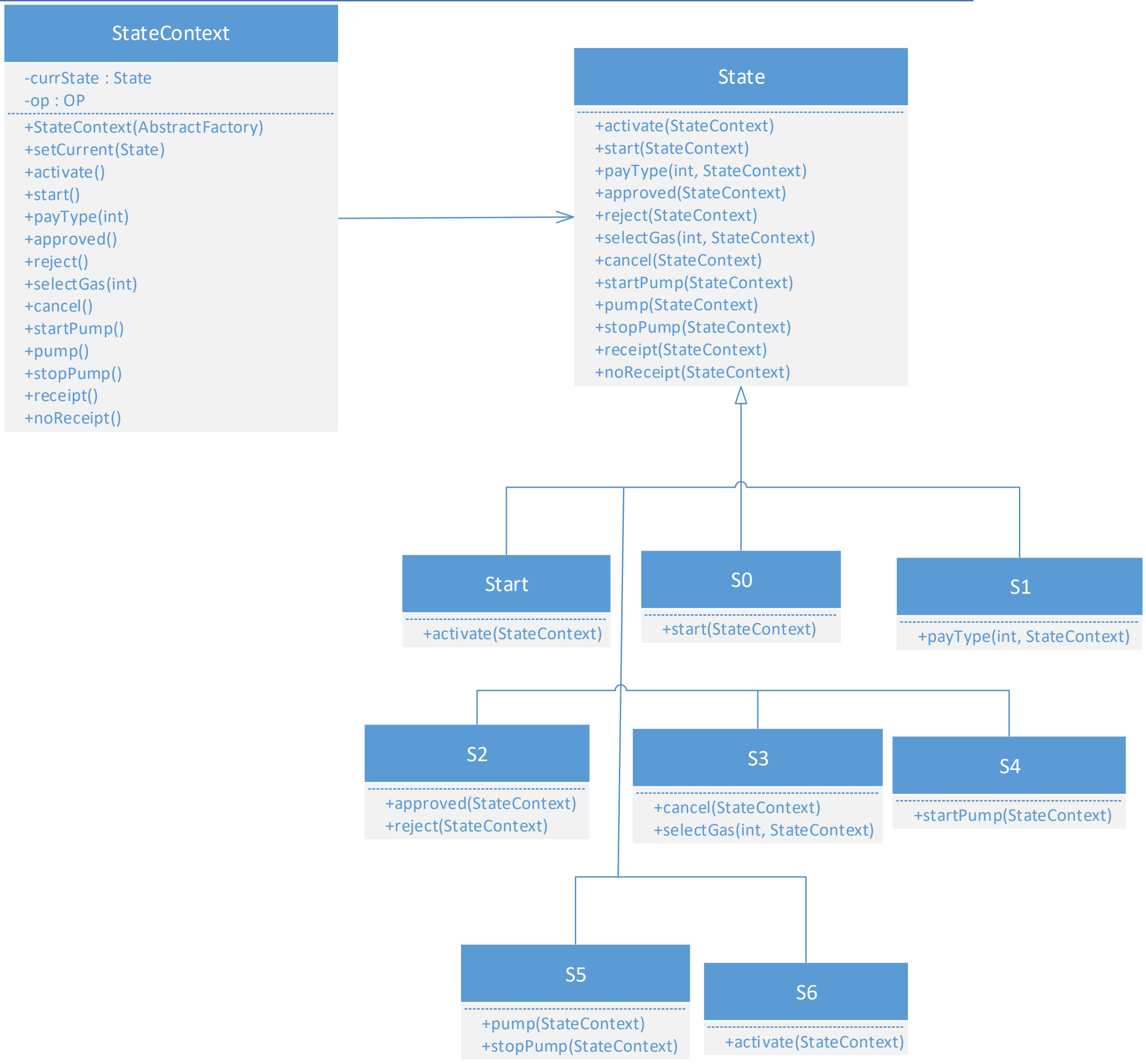
setSPriceF(float p)	Setter method to store spriceF value
getTotalF()	Getter method to return totalF value
setTotalF(float t)	Setter method to store totalF value
getGallon()	Getter method to return gallon value
setGallon(int g)	Setter method to store gallon value
getPriceF()	Getter method to return PriceF value
setPriceF(float p)	Setter method to store priceF value

❖ DataStore2

Purpose	This class represents the concrete implementation of Data Store which handles GasPump2 specific storage operations.
Member variables	tempRPriceI : temporary integer variable to store Regular gas type price tempSPriceI : temporary integer variable to store Super gas type price tempPPriceI : temporary integer variable to store Premium gas type price tempCash: temporarily float variable to store cash entered rPriceI : permanent integer variable to store Regular gas price sPriceI : permanent integer variable to store Super gas type price pPriceI : permanent integer variable to store Premium gas type price totalI : integer variable to store total value after gas has been pumped liter : integer variable to store liters pumped priceI : integer variable to store the price depending on the gas type chosen. cash: permanent float variable to store cash entered.
Member functions	
getTempRPriceI()	Getter method to return tempRPriceI value
setTempRPriceI(int p)	Setter method to store tempRpriceI value
getTempSPriceI()	Getter method to return tempSPriceI value
setTempPriceI(int p)	Setter method to store tempSpriceI value
getTempPPriceI()	Getter method to return tempPPriceI value
setTempPPriceI(int p)	Setter method to store tempPPriceI value
getTempCash()	Getter method to return tempCash value
setTempCash(int c)	Setter method to store tempCash value
getRPriceI()	Getter method to return rPriceI value
setRPriceI(int p)	Setter method to store rpriceI value
getSpriceI()	Getter method to return sPriceI value
setSPriceI(int p)	Setter method to store spriceI value
getTotalI()	Getter method to return totalI value
setTotalI(int t)	Setter method to store totalI value
getLiter()	Getter method to return liter value
setLiter(int l)	Setter method to store liter value
getPriceI()	Getter method to return PriceI value

setPriceI(int p)	Setter method to store priceI value
getCash()	Getter method to return cash value
setCash(float c)	Setter method to store cash value

3. *State Pattern*



❖ StateContext

Purpose	This class represents the context class which manages the states by a currState pointer as a reference to forward calls to specific state classes.
Member variables	currState: Pointer of State class to store current state op: Pointer of OP class to invoke actions from OP child classes
Member functions	
StateContext()	To initialize op object and point currState to Start state.
setCurrent(StateContext sc)	Change current state according to the state passed as argument

activate()	Call activate() of current state pointing to
start()	Call start () of current state pointing to
payType()	Call payType () of current state pointing to
approved()	Call approved () of current state pointing to
reject()	Call reject () of current state pointing to
cancel()	Call cancel () of current state pointing to
selectGas(int)	Call selectGas () of current state pointing to
startPump()	Call startPump() of current state pointing to
pump ()	Call pump () of current state pointing to
stopPump()	Call stopPump () of current state pointing to
receipt()	Call receipt () of current state pointing to
noReceipt()	Call noReceipt() of current state pointing to

- State

Purpose	This class represents the abstract class for States. Child state classes give the concrete implementation for all the functions in the current class.
Member variables	None
Member functions	
activate(StateContext sc)	Abstract methods. Logic implemented in child classes.
start(StateContext sc)	
payType(int t, StateContext sc)	
approved(StateContext sc)	
reject(StateContext sc)	
cancel(StateContext sc)	
selectGas(int g, StateContext sc)	
startPump(StateContext sc)	
pump (StateContext sc)	
stopPump(StateContext sc)	
receipt(StateContext sc)	
noReceipt(StateContext sc)	

- Start

Purpose	This class represents the concrete class for Start state.
Member variables	None
Member functions	
activate(StateContext sc)	Gas pump is activated by storing the data(prices of gas types) through OP class and sets the current state pointer to S0.

- S0

Purpose	This class represents the concrete class for S0 state.

Member variables	None
Member functions	
start(StateContext sc)	GasPump1: Pay Credit is displayed through OP class and sets the current state pointer to S1. GasPump2: Pay cash is displayed through OP class and sets the current state pointer to S1.

- S1

Purpose	This class represents the concrete class for S1 state.
Member variables	None
Member functions	
payType(int t, StateContext sc)	GasPump1: T argument has a value of 1 which suggests credit payment and sets the current state pointer to S2. GasPump2: T argument has a value of 2 which suggests cash payment stores the cash and display menu through OP class and sets the current state pointer to S3.

- S2

Purpose	This class represents the concrete class for S2 state.
Member variables	None
Member functions	
approved(StateContext sc)	GasPump1: Payment approved message is shown through OP class and sets the current state pointer to S3.
reject(StateContext sc)	GasPump1: Payment rejected message is shown through OP class and sets the current state pointer to S0.

- S3

Purpose	This class represents the concrete class for S3 state.
Member variables	None
Member functions	
cancel(StateContext sc)	Transaction Cancelled message is shown through OP class and current state pointer is sets the to S0.
selectGas(int g, StateContext sc)	Gas price is stored based on the gas type selected and current state pointers is set to S4.

- S4

Purpose	This class represents the concrete class for S4 state.
Member variables	None
Member functions	
startPump(StateContext sc)	Gas pump is started by initializing gas pump parameters and "Ready to pump" message is shown through OP class and current state pointer is set to S5.

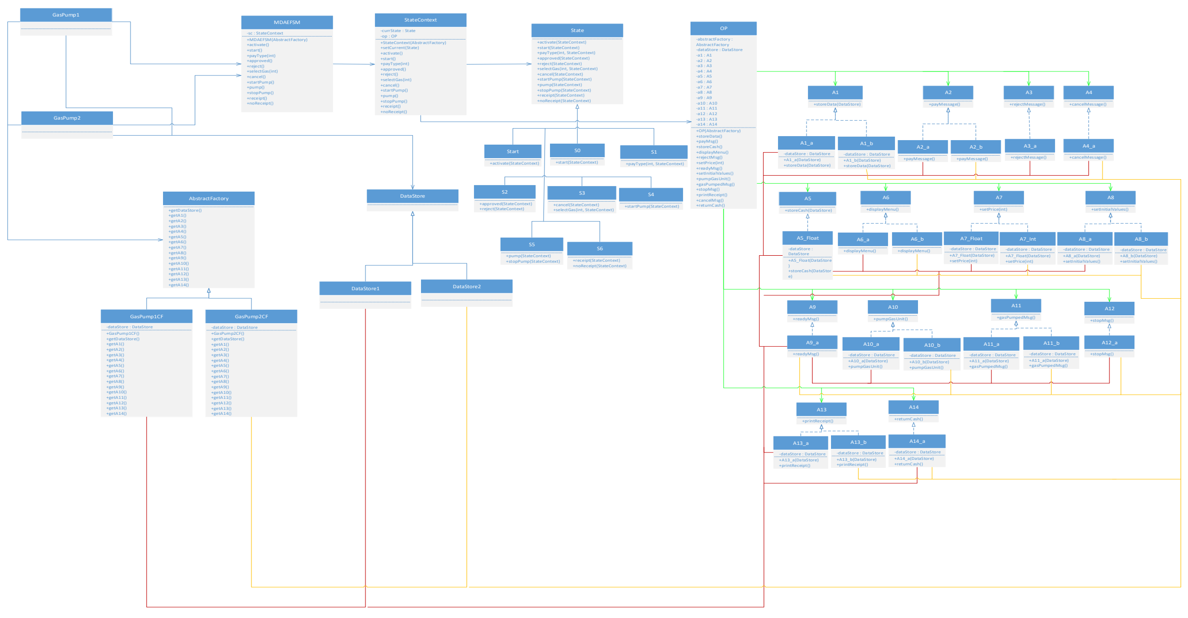
- S5

Purpose	This class represents the concrete class for S5 state.
Member variables	None
Member functions	
pump(StateContext sc)	A liter or gallon is pumped and gas units pumped is shown through OP class.
stopPump(StateContext sc)	Gas pump is stopped message is displayed and current state pointer is pointing to S6.

- S6

Purpose	This class represents the concrete class for S6 state.
Member variables	None
Member functions	
receipt(StateContext sc)	Receipt is generated for the gas disposed and cash left is returned depending on the gas pump through OP class and current state pointer is set to S0.
noReceipt(StateContext sc)	Only cash left is returned for gas pump 2 through OP class and current state pointer is set to S0.

4. *Abstract Factory Pattern*



Note: DataStore and GasPump classes member variables and functions have been removed on purpose to make the view better.

❖ AbstractFactory

Purpose	This class represents the abstract class for the factory method and is used to group all the concrete factories for GasPump1 and GasPump2.
Member variables	None
Member functions	
getDataStore()	Only Functions are declared here. Implementations for the functions are provided by child classes.
getA1()	
getA2()	
getA3()	
getA4()	
getA5()	
getA6()	
getA7()	
getA8()	
getA9()	
getA10()	
getA11()	
getA12()	
getA13()	
getA14()	

❖ GasPump1CF

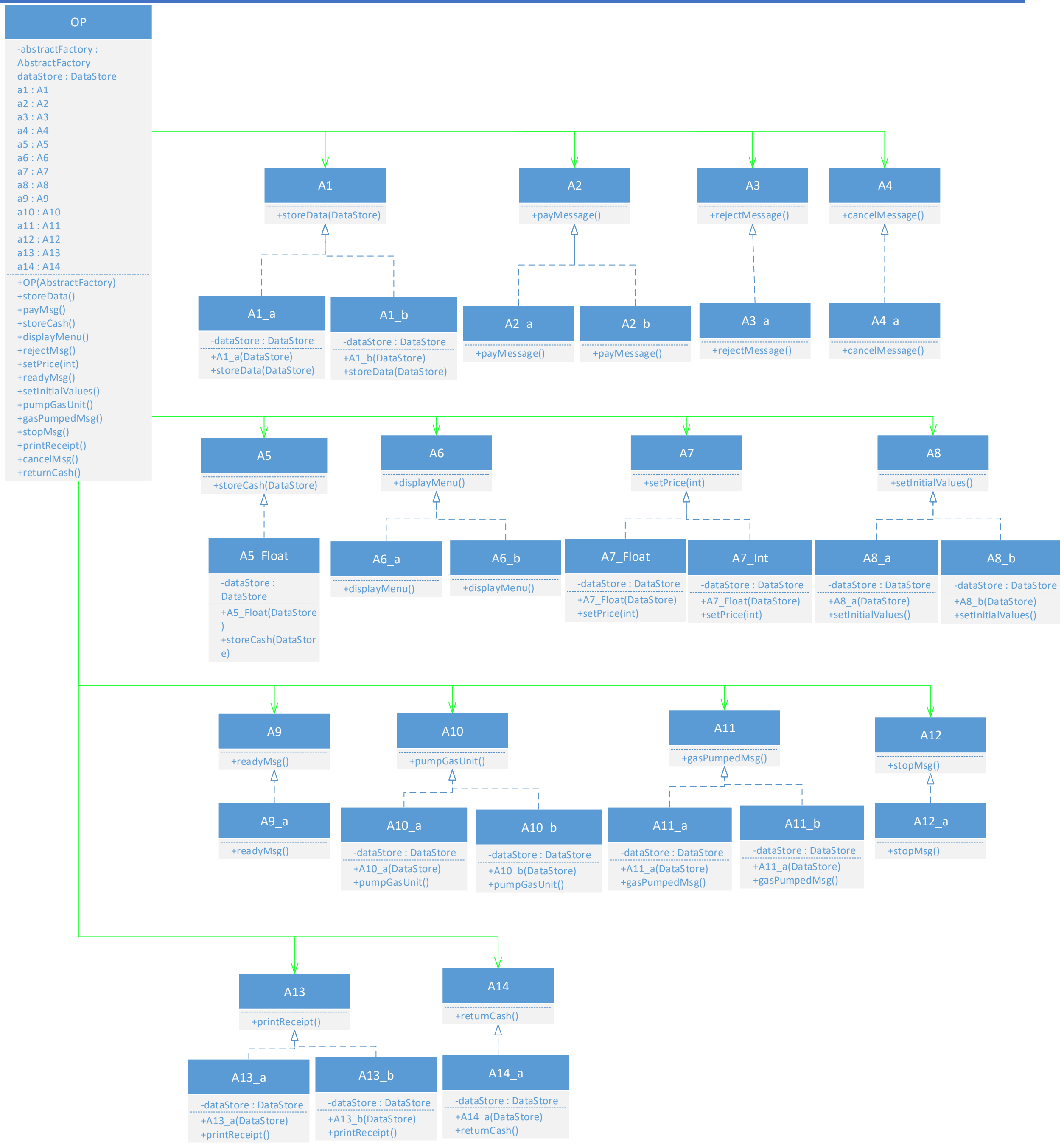
Purpose	This class represents the concrete factory class used to handle creation of objects relevant to GasPump1.
Member variables	dataStore is a pointer of DataStore class.
Member functions	
GasPump1CF()	Initialize dataStore to point to DataStore1 class
getDataStore()	Return the dataStore pointer
getA1()	Return a pointer to a new instance of A1_a(StoreData() specific to GasPump1)
getA2()	Return a pointer to a new instance of A2_a(payMessage() specific to GasPump1)
getA3()	Return a pointer to a new instance of A3_a(reject() method for GasPump1)
getA4()	Return a pointer to a new instance of A4_a(cancel() method for GasPump1)
getA5()	Return null since storeCash() method isn't applicable for GasPump2)
getA6()	Return a pointer to a new instance of A6_a(displayMenu() specific to GasPump1)
getA7()	Return a pointer to a new instance of A7_Float(setPrice() float version specific to GasPump1)
getA8()	Return a pointer to a new instance of A8_a(setInitialValues() specific to GasPump1)
getA9()	Return a pointer to a new instance of A9_a(readyMsg() method for GasPump1)
getA10()	Return a pointer to a new instance of A10_a(pumpGasUnit() specific to GasPump1)
getA11()	Return a pointer to a new instance of A11_a(gasPumpedMsg() specific to GasPump1)
getA12()	Return a pointer to a new instance of A12_a(stopMsg() method for GasPump1)
getA13()	Return a pointer to a new instance of A13_a(printReceipt() specific to GasPump1)

getA14()	Return a pointer to a new instance of A14_a(returnCash() specific to GasPump1)

❖ GasPump2CF

Purpose	This class represents the concrete factory class used to handle creation of objects relevant to GasPump2.
Member variables	dataStore is a pointer of DataStore class.
Member functions	
GasPump1CF()	Initialize dataStore to point to DataStore2 class
getDataStore()	Return the dataStore pointer
getA1()	Return a pointer to a new instance of A1_b(StoreData() specific to GasPump2)
getA2()	Return a pointer to a new instance of A2_b(payMessage() specific to GasPump2)
getA3()	Return a pointer to a new instance of A3_b(reject() method for GasPump2)
getA4()	Return a pointer to a new instance of A4_b(cancel() method for GasPump2)
getA5()	Return a pointer to a new instance of A5_Float(storeCash() method - float version for GasPump1)
getA6()	Return a pointer to a new instance of A6_b(displayMenu() specific to GasPump2)
getA7()	Return a pointer to a new instance of A7_Int(setPrice() integer version specific to GasPump2)
getA8()	Return a pointer to a new instance of A8_b(setInitialValues() specific to GasPump2)
getA9()	Return a pointer to a new instance of A9_b(readyMsg() method for GasPump2)
getA10()	Return a pointer to a new instance of A10_b(pumpGasUnit() specific to GasPump2)
getA11()	Return a pointer to a new instance of A11_b(gasPumpedMsg() specific to GasPump2)
getA12()	Return a pointer to a new instance of A12_b(stopMsg() method for GasPump2)
getA13()	Return a pointer to a new instance of A13_b(printReceipt() specific to GasPump2)
getA14()	Return a pointer to a new instance of A14_b(returnCash() specific to GasPump2)

5. *Strategy Pattern*



❖ OP

Purpose	This class represents the output processor for MDA and is the client of actions for various strategies. This contains multiple interfaces which are implemented by classes devising strategies.
Member variables	abstractFactory is a pointer of AbstractFactory responsible for invoking specific strategies based on the gas pump dataStore is a pointer of DataStore which provides access specific to each GasPump.

	a1 – a14 → objects of Action strategies interfaces used to invoke functions from specified strategy classes.
Member functions	
OP()	To initialize abstract factory, dataStore and action interface objects
storeData()	Call action from strategy a1
payMsg()	Call action from strategy a2
storeCash()	Call action from strategy a5
displayMenu()	Call action from strategy a6
rejectMsg()	Call action from strategy a3
setPrice(int g)	Call action from strategy a7
readyMsg(int)	Call action from strategy a9
setInitialValues()	Call action from strategy a8
pumpGasUnit()	Call action from strategy a10
gasPumpedMsg()	Call action from strategy a11
stopMsg()	Call action from strategy a12
printReceipt()	Call action from strategy a13
cancelMsg()	Call action from strategy a4
returnCash()	Call action from strategy a14

❖ A1

Purpose	This class represents the interface for various strategies of storeData() operation.
Member variables	None
Member functions	
storeData()	Abstract method.

❖ A1_a

Purpose	This class implements A1 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A1_a(DataStore ds)	To initialize dataStore object.
storeData()	Set Regular and Super gas price from the temporary variables.

❖ A1_b

Purpose	This class implements A1 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A1_b()	To initialize dataStore object.

storeData()	Set Regular and Super gas price from the temporary variables.

❖ A2

Purpose	This class represents the interface for various strategies of payMessage() operation.
Member variables	None
Member functions	
payMessage()	Abstract method.

❖ A2_a

Purpose	This class implements A2 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A2_a()	To initialize dataStore object.
payMessage()	Displays “Pay Credit” message

❖ A2_b

Purpose	This class implements A2 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A2_b()	To initialize dataStore object.
payMessage()	Displays “Pay Cash” message.

❖ A3

Purpose	This class represents the interface for various strategies of rejectMessage() operation.
Member variables	None
Member functions	
rejectMessage()	Abstract method.

❖ A3_a

Purpose	This class implements A3 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A3_a()	To initialize dataStore object.
rejectMessage()	Displays “Payment Rejected” message

- A4

Purpose	This class implements A4 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	None
Member functions	
cancelMessage()	Abstract method.

- A4_a

Purpose	This class implements A4 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A4_a()	To initialize dataStore object.
cancelMessage()	Displays “Transaction cancelled” message

- A5

Purpose	This class represents the interface for various strategies of storeCash() operation.
Member variables	None
Member functions	
storeCash()	Abstract method.

- A5_Float

Purpose	This class implements A5 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A5_Float(DataStore ds)	To initialize dataStore object.
storeCash()	Stores cash from the temporary variable

- A6

Purpose	This class represents the interface for various strategies of displayMenu() operation.
Member variables	None
Member functions	
displayMenu()	Abstract method.

- A6_a

Purpose	This class implements A6 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A6_a()	To initialize dataStore object.
displayMenu()	Display menu with respect to Gas Pump 1

❖ A6_b

Purpose	This class implements A6 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A6_b()	To initialize dataStore object.
displayMenu()	Display menu with respect to Gas Pump 2

❖ A7

Purpose	This class represents the interface for various strategies of setPrice() operation.
Member variables	None
Member functions	
setPrice(int g)	Abstract method.

❖ A7_Float

Purpose	This class implements A7 interface by overriding the abstract method with a specific strategy(float version) for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A7_Float()	To initialize dataStore object.
setPrice(int g)	Depending on the g value, stores the price(float) of regular/super gas price.

❖ A7_Int

Purpose	This class implements A7 interface by overriding the abstract method with a specific strategy(integer version) for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A7_Int()	To initialize dataStore object.
setPrice(int g)	Depending on the g value, stores the price(int) of regular/super gas price.

- A8

Purpose	This class represents the interface for various strategies of setInitialValues() operation.
Member variables	None
Member functions	
setInitialValues()	Abstract method.

- A8_a

Purpose	This class implements A8 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A8_a()	To initialize dataStore object.
setInitialValues()	Initializes gallons disposed and total to 0.

- A8_b

Purpose	This class implements A8 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A8_b()	To initialize dataStore object.
setInitialValues()	Initializes liters disposed and total to 0.

- A9

Purpose	This class represents the interface for various strategies of readyMsg() operation.
Member variables	None
Member functions	
readyMsg()	Abstract method.

- A9_a

Purpose	This class implements A9 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A9_a()	To initialize dataStore object.
readyMsg()	Set Regular and Super gas price from the temporary variables.

- A10

Purpose	This class represents the interface for various strategies of pumpGasUnit() operation.
Member variables	None
Member functions	
pumpGasUnit()	Abstract method.

❖ A10_a

Purpose	This class implements A10 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A10_a()	To initialize dataStore object.
pumpGasUnit()	Stores the updated gallons of gas disposed and respective total in the datastore.

❖ A10_b

Purpose	This class implements A10 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A10_b()	To initialize dataStore object.
pumpGasUnit()	Stores the updated liters of gas disposed and respective total in the datastore.

❖ A11

Purpose	This class represents the interface for various strategies of gasPumpedMsg() operation.
Member variables	None
Member functions	
gasPumpedMsg()	Abstract method.

❖ A11_a

Purpose	This class implements A11 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A11_a()	To initialize dataStore object.
gasPumpedMsg()	Display gas disposed in gallons.

❖ A11_b

Purpose	This class implements A11 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A11_b()	To initialize dataStore object.
gasPumpedMsg()	Display gas disposed in liters.

❖ A12

Purpose	This class represents the interface for various strategies of stopMsg() operation.
Member variables	None
Member functions	
stopMsg()	Abstract method.

❖ A12_a

Purpose	This class implements A12 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A12_a()	To initialize dataStore object.
stopMsg()	Display gas pump stopped message.

❖ A13

Purpose	This class represents the interface for various strategies of printReceipt() operation.
Member variables	None
Member functions	
printReceipt()	Abstract method.

❖ A13_a

Purpose	This class implements A13 interface by overriding the abstract method with a specific strategy for GasPump1.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A13_a()	To initialize dataStore object.
printReceipt()	Display total cost for the gas pumped.

❖ A13_b

Purpose	This class implements A13 interface by overriding the abstract method with a specific strategy for GasPump2.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A13_b()	To initialize dataStore object.
printReceipt()	Display liters pumped and respective total cost.

❖ A14

Purpose	This class represents the interface for various strategies of returnCash() operation.
Member variables	None
Member functions	
returnCash()	Abstract method.

❖ A14_a

Purpose	This class implements A14 interface by overriding the abstract method with a generic strategy for both the GasPumps.
Member variables	dataStore is a pointer object of DataStore class which helps in accessing data specific to each gas pump through factory object.
Member functions	
A14_a()	To initialize dataStore object.
returnCash()	Returns the remaining cash.

Pattern description:

State Pattern

No. of classes – 9

Classes – StateContext, State, Start, S0, S1, S2, S3, S4, S5, S6.

Abstract Factory Pattern

No. of classes – 3

Classes – AbstractFactory, GasPump1CF, GasPump2CF

Strategy pattern

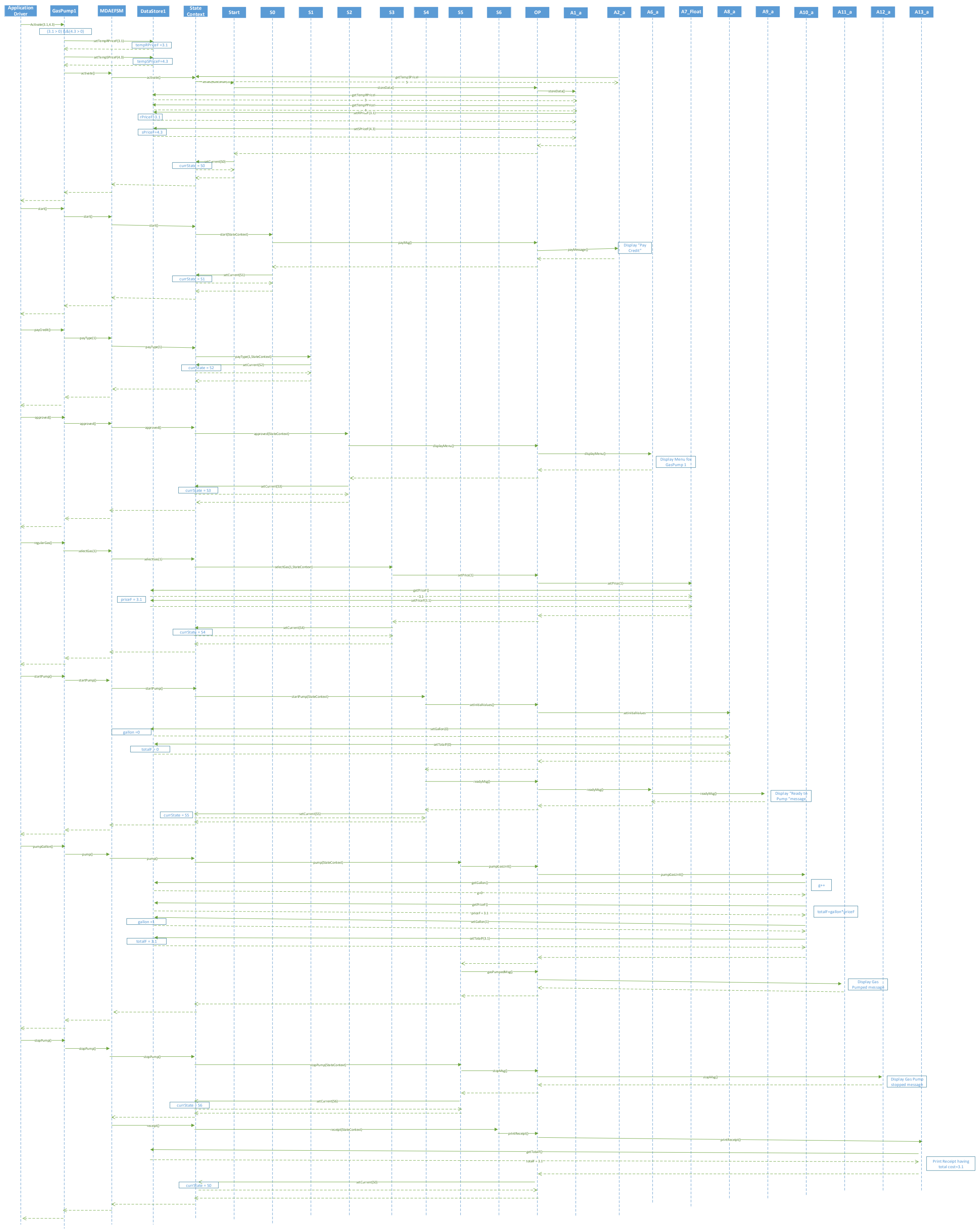
No. of classes – 36

Abstract classes – A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14

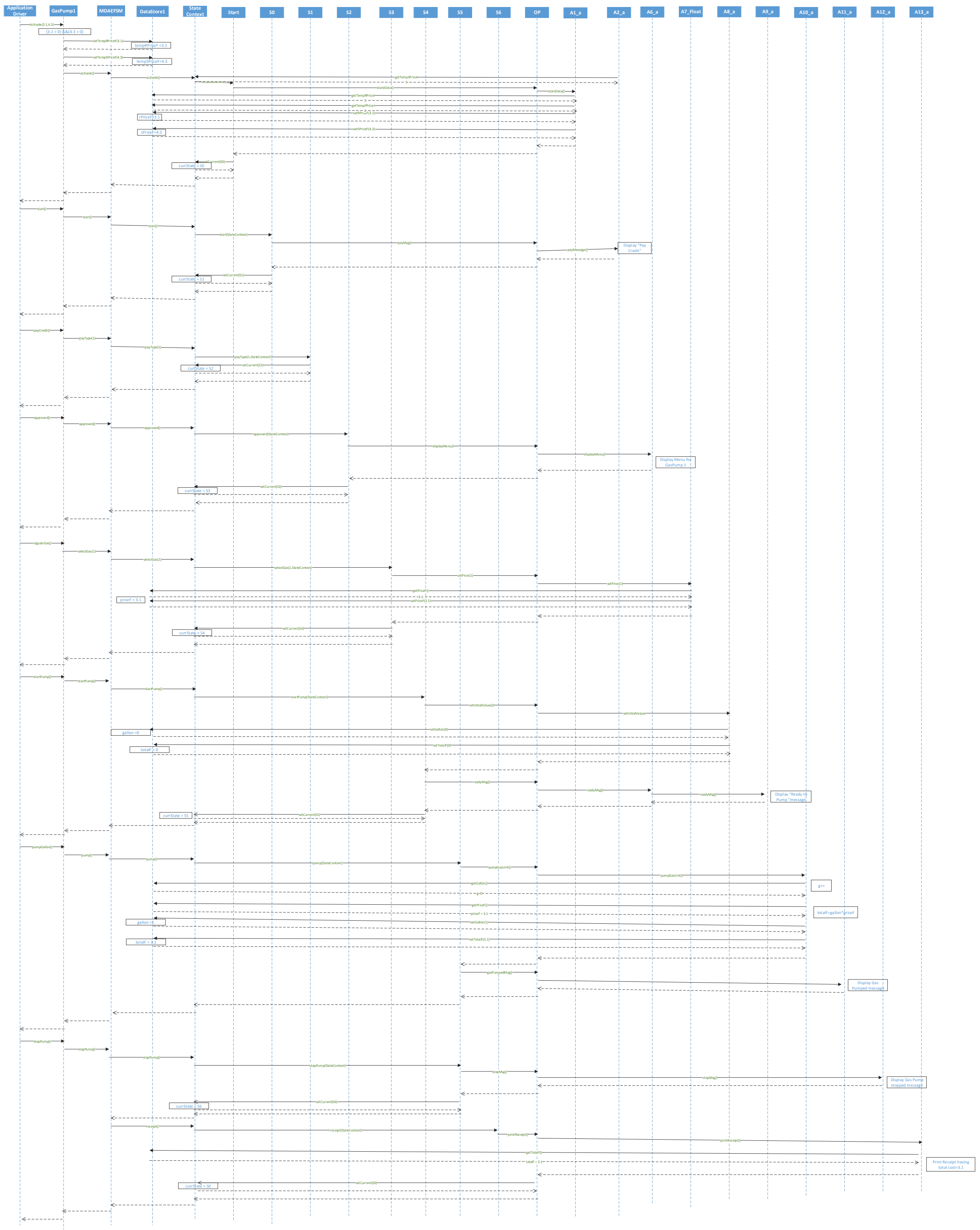
Concrete classes – A1_a, A1_b, A2_a, A2_b, A3_a, A4_a, A5_Float, A6_a, A6_b, A7_Float, A7_Int, A8_a, A8_b, A9_a, A10_a,A10_b,A11_a, A11_b,A12_a,A13_a,A13_b,A14_a

Sequence Diagram:

Scenario – 1: *Activate(3.1, 4.3), Start(), PayCredit(), Approved(), Regular(), StartPump(), PumpGallon(),StopPump()*



Scenario -2: *Activate(3, 4, 5), Start(), PayCash(6), Premium(),StartPump(), PumpLiter(), PumpLiter(), NoReceipt()*



Source code:

1. ApplicationDriver.java

```
package application;
import java.util.Scanner;

/**
 * This class is the trigger point when the jar is executed.
 * This class manages the user interface, interprets the choices to invoke desired events and actions.
 * @author cheth
 *
 */
public class ApplicationDriver {

	public static void main(String args[]){
		System.out.println("\n##########################");
		System.out.println("Welcome To Gas Pump System!");
		System.out.println("##########################");
		String choice = "";

		while(!choice.equals("Q")){

			System.out.println("1.Gas pump-1\t2.Gas Pump-2\tQ.Exit\nMake Your Choice: ");
			Scanner input = new Scanner(System.in);
			choice = input.next();

			switch(choice){
				case "1": System.out.println("\n****************Gas Pump 1****************");
						printBorder();
						System.out.println("Menu options are shown once after 
choosing gas system. Please note down the choices.");
						System.out.println("0.Activate\n1.Start\t2.Pay 
Credit\t3.Reject\t4.Cancel\t5.Approved\n6.Super\t7.Regular\t"
								+ "8.Start Pump\t9.Pump 
Gallon\t10.Stop Pump\nQ.Exit Driver");
						printBorder();
						gasPump1Driver();
						break;
				case "2": System.out.println("****************Gas Pump 2****************");
						printBorder();
						System.out.println("Menu options are shown once after 
choosing gas system. Please note down the choices.");
						System.out.println("0.Activate\n1.Start\t2.Pay 
Cash\t3.Cancel\t4.Premium\t5.Regular\n6.Super\t7.Start Pump\t"
								+ "8.Pump 
Liter\t9.Stop\t10.Receipt\t11.No Receipt\nQ.Exit Driver");
						printBorder();
						gasPump2Driver();
						break;
				case "Q":System.out.println("Thank you.Please Visit Again!!");
						break;
				default: System.out.println("Invalid choice");
						break;
			}
		}
	}

	public static void gasPump1Driver(){
		String choice = "";
		GasPump1 gp1 = new GasPump1();

		while(!choice.equals("Q")){
			System.out.print("Please make a choice? ");

			Scanner input = new Scanner(System.in);
```

```
                        choice = input.next();

                        switch(choice){
                            case "0": System.out.println("Executing Activate(float a, float b)");
                                        System.out.print("Enter numerical value for a: ");
                                        float a = input.nextFloat();
                                        System.out.print("Enter numerical value for b: ");
                                        float b = input.nextFloat();
                                        gp1.activate(a, b);
                                        break;
                            case "1": System.out.println("\nExecuting Start()..\n");
                                        gp1.start();
                                    break;
                            case "2": System.out.println("\nExecuting PayCredit()..\n");
                                        gp1.payCredit();
                                    break;
                            case "3": System.out.println("\nExecuting Reject()..\n");
                                        gp1.reject();
                                        break;
                            case "4": System.out.println("\nExecuting Cancel()..\n");
                                        gp1.cancel();
                                        break;
                            case "5": System.out.println("\nExecuting Approved()..\n");
                                        gp1.approved();
                                        break;
                            case "6": System.out.println("\nExecuting Super()..\n");
                                        gp1.superGas();
                                        break;
                            case "7": System.out.println("\nExecuting Regular()..\n");
                                        gp1.regularGas();
                                        break;
                            case "8": System.out.println("\nExecuting StartPump()..\n");
                                        gp1.startPump();
                                        break;
                            case "9": System.out.println("\nExecuting PumpGallon()..\n");
                                        gp1.pumpGallon();
                                        break;
                            case "10": System.out.println("\nExecuting StopPump()..\n");
                                        gp1.stopPump();
                                        break;
                            case "Q": System.out.println("\nExiting driver..\n");
                                    break;
                            default: System.out.println("\nInvalid choice\n");
                                    break;
                        }
                }
//              input.close();
                gp1 = null;
        }


        public static void gasPump2Driver(){
                String choice = "";
                GasPump2 gp2 = new GasPump2();

                while(!choice.equals("Q")){
                        System.out.print("Please make a choice? ");

                        Scanner input = new Scanner(System.in);
                        choice = input.next();

                        switch(choice){
                            case "0": System.out.println("Executing Activate(int a, int b,int c)");
                                        System.out.print("Enter numerical value for a: ");
                                        int a = input.nextInt();
                                        System.out.print("Enter numerical value for b: ");
                                        int b = input.nextInt();
                                        System.out.print("Enter a numerical value for c: ");
```

```
                                                        int c = input.nextInt();
                                                        gp2.activate(a, b, c);
                                                        break;
                                    case "1": printBorder();
                                                        System.out.println("\nExecuting Start()..\n");
                                                        gp2.start();
                                                  break;
                                    case "2": printBorder();
                                                        System.out.println("\nExecuting PayCash()..\n");
                                                        System.out.print("Enter cash amount: ");
                                                        float cash = input.nextFloat();
                                                        gp2.payCash(cash);
                                                  break;
                                    case "3": System.out.println("\nExecuting Cancel()..\n");
                                                        gp2.cancel();
                                                        break;
                                    case "4": System.out.println("\nExecuting Premium()..\n");
                                                        gp2.premiumGas();
                                                        break;
                                    case "5": System.out.println("\nExecuting Regular()..\n");
                                                        gp2.regularGas();
                                                        break;
                                    case "6": System.out.println("\nExecuting Super()..\n");
                                                        gp2.superGas();
                                                        break;
                                    case "7": System.out.println("\nExecuting StartPump()..\n");
                                                        gp2.startPump();
                                                        break;
                                    case "8": System.out.println("\nExecuting PumpLiter()..\n");
                                                        gp2.pumpLiter();
                                                        break;
                                    case "9": System.out.println("\nExecuting Stop()..\n");
                                                        gp2.stop();
                                                        break;
                                    case "10": System.out.println("\nExecuting Receipt()..\n");
                                                        gp2.receipt();
                                                        break;
                                    case "11": System.out.println("\nExecuting NoReceipt()..\n");
                                                        gp2.noReceipt();
                                                        break;
                                    case "Q": System.out.println("\nExiting driver..\n");
                                                 break;
                                    default: System.out.println("\nInvalid choice\n");
                                                break;
                              }
                        }
//                      input.close();
                        gp2 = null;
      }

      public static void printBorder(){

      System.out.println("============================================================================
================");
      }
}
```

2. GasPump1.java

```
      package application;

      import abstractfactory.AbstractFactory;
      import abstractfactory.GasPump1CF;
      import datastore.DataStore;
      /**
       * This class represents GasPump1 component Implementation which addresses all the needs of GasPump1
      operations.
```

```
 * @author cheth
 *
 */
public class GasPump1 {
	MDAEFSM mdaEfsm;
	AbstractFactory factory;
	DataStore dataStore;
	/*
	 * Constructor to initialize abstract factory, datastore and mdaefsm objects
	 */
	public GasPump1() {
		factory = new GasPump1CF();
		mdaEfsm = new MDAEFSM(factory);
		dataStore = factory.getDataStore();
	}
	/*
	 * Function to activate the gas pump with initial parameters used Function to set gas cost
	 */
	public void activate(float a,float b){
		if(a>0 && b>0){
			dataStore.setTempRPriceF(a);
			dataStore.setTempSPriceF(b);
			mdaEfsm.activate();
		}
	}
	/*
	 * Function to start the gas pump component
	 */
	public void start(){
		mdaEfsm.start();
	}
	/*
	 * Function to make the payment for gas through credit card.
	 */
	public void payCredit(){
		mdaEfsm.payType(1);
	}
	/*
	 * Function to approve the credit card.
	 */
	public void approved(){
		mdaEfsm.approved();
	}
	/*
	 * Function to reject the credit card payment.
	 */
	public void reject(){
		mdaEfsm.reject();
	}
	/*
	 * Function to select Super gas type
	 */
	public void superGas(){
		mdaEfsm.selectGas(2);
	}
	/*
	 * Function to select Regular gas type
	 */
	public void regularGas(){
		mdaEfsm.selectGas(1);
	}
	/*
	 * Function to cancel the transaction
	 */
	public void cancel(){
		mdaEfsm.cancel();
	}
	/*
```

```
		 * Function to start pumping gas
		 */
		public void startPump(){
			mdaEfsm.startPump();
		}
		/*
		 * Function to dispose a gallon of gas
		 */
		public void pumpGallon(){
			mdaEfsm.pump();
		}
		/*
		 * Function to stop pump gas
		 */
		public void stopPump(){
			mdaEfsm.stopPump();
			mdaEfsm.receipt();
		}

	}
```

3. GasPump2.java

```
package application;

import abstractfactory.AbstractFactory;
import abstractfactory.GasPump2CF;
import datastore.DataStore;
/**
 * This class represents GasPump2 component Implementation which addresses all the needs of GasPump2 operations.
 * @author cheth
 *
 */
public class GasPump2 {

	MDAEFSM mdaEfsm;
	AbstractFactory factory;
	DataStore dataStore;
	/*
	 * Constructor to initialize abstract factory, datastore and mdaefsm objects.
	 */
	public GasPump2() {
		factory = new GasPump2CF();
		mdaEfsm = new MDAEFSM(factory);
		dataStore = factory.getDataStore();
	}
	/*
	 * Function to activate the gas pump with initial parameters used Function to set gas cost
	 */
	public void activate(int a,int b, int c){
		if(a>0 && b>0 && c>0){
			dataStore.setTempRPriceI(a);
			dataStore.setTempPPriceI(b);
			dataStore.setTempSPriceI(c);
			mdaEfsm.activate();
		}
	}

	/*
	 * Function to start the gas pump component
	 */
	public void start(){
		mdaEfsm.start();
	}
	/*
	 * Function to make the payment for gas through cash where parameter represents cash entered.
	 */
	public void payCash(float c){
```

```
                if(c>0){
                        dataStore.setTempCash(c);
                        mdaEfsm.payType(2);
                }
        }
        /*
         * Function to select Super gas type
         */
        public void superGas(){
                mdaEfsm.selectGas(2);
        }
        /*
         * Function to select Premium gas type
         */
        public void premiumGas(){
                mdaEfsm.selectGas(3);
        }
        /*
         * Function to select Regular gas type
         */
        public void regularGas(){
                mdaEfsm.selectGas(1);
        }
        /*
         * Function to cancel the transaction
         */
        public void cancel(){
                mdaEfsm.cancel();
        }
        /*
         * Function to start pumping gas
         */
        public void startPump(){
                mdaEfsm.startPump();
        }
        /*
         * Function to dispose a liter of gas
         */
        public void pumpLiter(){
                float currentGasCost = (dataStore.getLiter() + 1)*dataStore.getPriceI();
                if(dataStore.getCash() < currentGasCost){
                        mdaEfsm.stopPump();
                } else {
                        mdaEfsm.pump();
                }
        }
        /*
         * Function to stop pump gas
         */
        public void stop(){
                mdaEfsm.stopPump();
        }
        /*
         * Function to request a receipt
         */
        public void receipt(){
                mdaEfsm.receipt();
        }
        /*
         * Function to not provide a receipt
         */
        public void noReceipt(){
                mdaEfsm.noReceipt();
        }
}
```

4. MDAEFSM.java

```
package application;
```

```
import abstractfactory.AbstractFactory;
import state.StateContext;

/**
 * This class represents platform independent logic for all its clients i.e.,GasPump1, GasPump2.
 * Seperation of this functionality allows for reduced effort during maintenance and when new clients are added.
 * @author cheth
 *
 */
public class MDAEFSM {

	StateContext stateCtx;
	/*
	 * To initialize sc object by passing factory object
	 */
	public MDAEFSM(AbstractFactory factory) {
		stateCtx = new StateContext(factory);
	}
	/*
	 * Call activate() through sc of StateContext class
	 */
	public void activate(){
		stateCtx.activate();
	}
	/*
	 * Call start() through sc of StateContext class
	 */
	public void start(){
		stateCtx.start();
	}
	/*
	 * Call payType() through sc of StateContext class
	 */
	public void payType(int t){
		stateCtx.payType(t);
	}
	/*
	 * Call approved() through sc of StateContext class
	 */
	public void approved(){
		stateCtx.approved();
	}
	/*
	 * Call reject() through sc of StateContext class
	 */
	public void reject(){
		stateCtx.reject();
	}
	/*
	 * Call selectGas() through sc of StateContext class
	 */
	public void selectGas(int g){
		stateCtx.selectGas(g);
	}
	/*
	 * Call cancel() through sc of StateContext class
	 */
	public void cancel(){
		stateCtx.cancel();
	}
	/*
	 * Call startPump() through sc of StateContext class
	 */
	public void startPump(){
		stateCtx.startPump();
	}
	/*
	 * Call pump() through sc of StateContext class
	 */
	public void pump(){
		stateCtx.pump();
	}
	/*
	 * Call stopPump() through sc of StateContext class
	 */
	public void stopPump(){
		stateCtx.stopPump();
	}
	/*
	 * Call receipt() through sc of StateContext class
	 */
	public void receipt(){
		stateCtx.receipt();
	}
	/*
	 * Call noReceipt() through sc of StateContext class
	 */
	public void noReceipt(){
		stateCtx.noReceipt();
	}
}
```

5. AbstractFactory.java

```
package abstractfactory;

import datastore.DataStore;
import strategy.*;

/*
 * This class represents the abstract class for the factory method and  is used to group all the concrete factories for
GasPump1 and GasPump2.
 */
public abstract class AbstractFactory {
	/* Below are abstract methods*/
	public abstract DataStore getDataStore();
	public abstract A1 getA1();
	public abstract A2 getA2();
	public abstract A3 getA3();
	public abstract A4 getA4();
	public abstract A5 getA5();
	public abstract A6 getA6();
	public abstract A7 getA7();
	public abstract A8 getA8();
	public abstract A9 getA9();
	public abstract A10 getA10();
	public abstract A11 getA11();
	public abstract A12 getA12();
	public abstract A13 getA13();
	public abstract A14 getA14();
}
```

6. GasPump1CF.java

```
package abstractfactory;

import datastore.DataStore;
import datastore.DataStore1;
import strategy.*;

/**
 * This class represents the concrete factory class used to handle creation of objects relevant to GasPump1.
 * @author cheth
 *
 */
public class GasPump1CF extends AbstractFactory{

	DataStore dataStore;
	/*
 * Initialize dataStore to point to DataStore1 class
 */
public GasPump1CF() {
		dataStore = new DataStore1();
}
/*
 * Return a pointer to a new instance of A1_a(StoreData() specific to GasPump1)
 */
public DataStore getDataStore() {
		return dataStore;
}
/*
 * Return a pointer to a new instance of A2_a(payMessage() specific to GasPump1)
 */
public A1 getA1() {
		return new A1_a(dataStore);
}
/*
 * Return a pointer to a new instance of A2_a(payMessage() specific to GasPump1)
 */
public A2 getA2() {
		return new A2_a();
}
/*
 * Return a pointer to a new instance of A3_a(reject() method for GasPump1)
 */
public A3 getA3() {
		return new A3_a();
}
/*
 * Return a pointer to a new instance of A4_a(cancel() method for GasPump1)
 */
public A4 getA4() {
		return new A4_a();
}
/*
 * Return null since storeCash() method isn't applicable for GasPump2)
 */
public A5 getA5() {
		return null;//return new A5_Float(dataStore);//Not used
}
/*
 * Return a pointer to a new instance of A6_a(displayMenu() specific to GasPump1)
 */
public A6 getA6() {
		return new A6_a();
}
/*
 * Return a pointer to a new instance of A7_Float(setPrice() float version specific to GasPump1)
 */
public A7 getA7() {
		return new A7_Float(dataStore);
}
/*
 * Return a pointer to a new instance of A8_a(setInitialValues() specific to GasPump1)
 */
public A8 getA8() {
		return new A8_a(dataStore);
}
/*
 * Return a pointer to a new instance of A9_a(readyMsg() method for GasPump1)
 */
public A9 getA9() {
		return new A9_a();
}
/*
 * Return a pointer to a new instance of A10_a(pumpGasUnit() specific to GasPump1)
 */
	public A10 getA10() {
		return new A10_a(dataStore);
	}
	/*
	 * Return a pointer to a new instance of A11_a(gasPumpedMsg() specific to GasPump1)
	 */
	public A11 getA11() {
		return new A11_a(dataStore);
	}
	/*
	 * Return a pointer to a new instance of A12_a(stopMsg() method for GasPump1)
	 */
	public A12 getA12() {
		return new A12_a();
	}
	/*
	 * Return a pointer to a new instance of A13_a(printReceipt() specific to GasPump1)
	 */
	public A13 getA13() {
		return new A13_a(dataStore);
	}
	/*
	 * Return a pointer to a new instance of A14_a(returnCash() specific to GasPump1)
	 */
	public A14 getA14() {
		return new A14_a(dataStore);
	}

}
```

7. GasPump2CF.java

```
package abstractfactory;

import datastore.DataStore;
import datastore.DataStore2;
import strategy.*;

/**
 * This class represents the concrete factory class used to handle creation of objects relevant to GasPump2.
 * @author cheth
 *
 */
public class GasPump2CF extends AbstractFactory{

	DataStore dataStore;
	/*
	 * Initialize dataStore to point to DataStore2 class
	 */
	public GasPump2CF() {
		dataStore = new DataStore2();
	}
	/*
	 * Return the dataStore pointer
	 */
	public DataStore getDataStore() {
		return dataStore;
	}
	/*
	 * Return a pointer to a new instance of A1_b(StoreData() specific to GasPump2)
	 */
	public A1 getA1() {
		return new A1_b(dataStore);
	}
	/*
	 * Return a pointer to a new instance of A2_b(payMessage() specific to GasPump2)
	 */
	public A2 getA2() {
		return new A2_b();
}
/*
 * Return a pointer to a new instance of A3_b(reject() method for GasPump2)
 */
public A3 getA3() {
	return null;//new A3_a();
}
/*
 * Return a pointer to a new instance of A4_b(cancel() method for GasPump2)
 */
public A4 getA4() {
	return new A4_a();
}
/*
 * Return a pointer to a new instance of A5_Float(storeCash() method - float version for GasPump1)
 */
public A5 getA5() {
	return new A5_Float(dataStore);
}
/*
 * Return a pointer to a new instance of A6_b(displayMenu() specific to GasPump2)
 */
public A6 getA6() {
	return new A6_b();
}
/*
 * Return a pointer to a new instance of A7_Int(setPrice() integer version specific to GasPump2)
 */
public A7 getA7() {
	return new A7_Int(dataStore);
}
/*
 * Return a pointer to a new instance of A8_b(setInitialValues() specific to GasPump2)
 */
public A8 getA8() {
	return new A8_b(dataStore);
}
/*
 * Return a pointer to a new instance of A9_b(readyMsg() method for GasPump2)
 */
public A9 getA9() {
	return new A9_a();
}

/*
 * Return a pointer to a new instance of A10_b(pumpGasUnit() specific to GasPump2)
 */
public A10 getA10() {
	return new A10_b(dataStore);
}

/*
 * Return a pointer to a new instance of A11_b(gasPumpedMsg() specific to GasPump2)
 */
public A11 getA11() {
	return new A11_b(dataStore);
}

/*
 * Return a pointer to a new instance of A12_b(stopMsg() method for GasPump2)
 */
public A12 getA12() {
	return new A12_a();
}

/*
 * Return a pointer to a new instance of A13_b(printReceipt() specific to GasPump2)
 */
```

```
		public A13 getA13() {
			return new A13_b(dataStore);
		}

		/*
		 * Return a pointer to a new instance of A14_b(returnCash() specific to GasPump2)
		 */
		public A14 getA14() {
			return new A14_a(dataStore);
		}

	}
```

8. DataStore.java

```
package datastore;

/**
 * This class represents the abstract class for the Data Store which
 * is used to group all the various data store concrete classes within the implementation.
 * @author cheth
 *
 */
public abstract class DataStore {

	/*
	 * Methods below are abstract and child classes have the implementation.
	 */
	//for GP1
	public float getTempRPriceF(){ return 0; }

	public void setTempRPriceF(float f){}

	public float getTempSPriceF(){ return 0; }

	public void setTempSPriceF(float f){}

	public float getRPriceF() { return 0; }

	public void setRPriceF(float rPrice) {}

	public float getSPriceF() { return 0; }

	public void setSPriceF(float sPrice) {}

	public float getTotalF() { return 0; }

	public void setTotalF(float total) {}

	public int getGallon() { return 0; }

	public void setGallon(int g) {}

	public float getPriceF(){ return 0; }

	public void setPriceF(float price){}


	//for GP2
	public int getTempRPriceI(){ return 0; }

	public void setTempRPriceI(int i){}

	public int getTempSPriceI(){ return 0; }

	public void setTempSPriceI(int i){}

	public int getTempPPriceI(){ return 0; }
	public void setTempPPriceI(int i){}

	public int getRPriceI() { return 0; }

	public void setRPriceI(int rPrice) {}

	public int getSPriceI() { return 0; }

	public void setSPriceI(int sPrice) {}

	public int getPPriceI() { return 0; }

	public void setPPriceI(int pPrice) {}

	public float getTempCash(){ return 0; }

	public void setTempCash(float cash){}

	public float getCash() { return 0; }

	public void setCash(float cash) {}

	public int getPriceI(){ return 0; }

	public void setPriceI(int price){}

	public int getLiter(){ return 0; }

	public void setLiter(int l){}

	public int getTotalI() { return 0; }

	public void setTotalI(int price) {}
}
```

9. DataStore1.java

```
package datastore;

/**
 * This class represents the concrete implementation of Data Store which handles GasPump1 specific storage
operations.
 * @author cheth
 *
 */
public class DataStore1 extends DataStore {

	private float tempRPriceF;
	private float tempSPriceF;
	private float rPriceF;
	private float sPriceF;
	private float totalF;
	private int gallon;
	private float priceF;

	/*
	 * Getter method to return tempRPriceF value
	 */
	public float getTempRPriceF() {
		return tempRPriceF;
	}

	/*
	 * Setter method to store tempRpriceF value
	 * @see datastore.DataStore#setTempRPriceF(float)
	 */
	public void setTempRPriceF(float tempRPriceF) {
		this.tempRPriceF = tempRPriceF;
	}
	/*
	 * Getter method to return tempSPriceF value
	 * @see datastore.DataStore#getTempSPriceF()
	 */
	public float getTempSPriceF() {
		return tempSPriceF;
	}
	/*
	 * (non-Javadoc)
	 * @see datastore.DataStore#setTempSPriceF(float)
	 */
	public void setTempSPriceF(float tempSPriceF) {
		this.tempSPriceF = tempSPriceF;
	}
	/*
	 * Getter method to return rPriceF value
	 * @see datastore.DataStore#getRPriceF()
	 */
	public float getRPriceF() {
		return rPriceF;
	}
	/*
	 * Setter method to store rpriceF value
	 * @see datastore.DataStore#setRPriceF(float)
	 */
	public void setRPriceF(float rPriceF) {
		this.rPriceF = rPriceF;
	}
	/*
	 * Getter method to return sPriceF value
	 * @see datastore.DataStore#getSPriceF()
	 */
	public float getSPriceF() {
		return sPriceF;
	}
	/*
	 * Setter method to store spriceF value
	 * @see datastore.DataStore#setSPriceF(float)
	 */
	public void setSPriceF(float sPriceF) {
		this.sPriceF = sPriceF;
	}
	/*
	 * Getter method to return totalF value
	 * @see datastore.DataStore#getTotalF()
	 */
	public float getTotalF() {
		return totalF;
	}
	/*
	 *Setter method to store totalF value
	 * @see datastore.DataStore#setTotalF(float)
	 */
	public void setTotalF(float total) {
		this.totalF = total;
	}
	/*
	 * Getter method to return gallon value
	 * @see datastore.DataStore#getGallon()
	 */
	public int getGallon() {
		return gallon;
	}
	/*
	 * Setter method to store gallon value
	 * @see datastore.DataStore#setGallon(int)
	 */
	public void setGallon(int gallon) {
		this.gallon = gallon;
	}
	/*
	 * Getter method to return PriceF value
	 * @see datastore.DataStore#getPriceF()
	 */
	public float getPriceF() {
		return priceF;
	}
	/*
	 * Setter method to store priceF value
	 * @see datastore.DataStore#setPriceF(float)
	 */
	public void setPriceF(float priceF) {
		this.priceF = priceF;
	}

}
```

10. DataStore2.java

```
package datastore;
/**
 * This class represents the concrete implementation of Data Store which handles GasPump2 specific storage 
operations.
 * @author cheth
 *
 */
public class DataStore2 extends DataStore {

	private int tempRPriceI;
	private int tempSPriceI;
	private int tempPPriceI;
	private int rPriceI;
	private int sPriceI;
	private int pPriceI;
	private float tempCash;
	private float cash;
	private int priceI;
	private int liter;
	private int totalI;
	/*
	 * Getter method to return tempRPriceI value
	 */
	public int getTempRPriceI() {
		return tempRPriceI;
	}
	/*
	 * Setter method to store tempSpriceI value
	 */
	public void setTempRPriceI(int tempRPriceI) {
		this.tempRPriceI = tempRPriceI;
	}
	/*
	 * Getter method to return tempSPriceI value
	 */
	public int getTempSPriceI() {
		return tempSPriceI;
	}
	/*
	 *
	 */
	public void setTempSPriceI(int tempSPriceI) {
		this.tempSPriceI = tempSPriceI;
	}
	/*
 * Getter method to return tempPPriceI value
 */
public int getTempPPriceI() {
		return tempPPriceI;
}
/*
 * Setter method to store tempPPriceI value
 */
public void setTempPPriceI(int tempPPriceI) {
		this.tempPPriceI = tempPPriceI;
}
/*
 * Getter method to return rPriceI value
 */
public int getRPriceI() {
		return rPriceI;
}
/*
 * Setter method to store rpriceI value
 */
public void setRPriceI(int rPriceI) {
		this.rPriceI = rPriceI;
}
/*
 * Getter method to return sPriceI value
 */
public int getSPriceI() {
		return sPriceI;
}
/*
 * Setter method to store spriceI value
 */
public void setSPriceI(int sPriceI) {
		this.sPriceI = sPriceI;
}
/*
 * Getter method to return tempPPriceI value
 */
public int getPPriceI() {
		return pPriceI;
}
/*
 * Setter method to store tempPPriceI value
 */
public void setPPriceI(int pPriceI) {
		this.pPriceI = pPriceI;
}
/*
 * Getter method to return tempCash value
 */
public float getTempCash() {
		return tempCash;
}
/*
 * Setter method to store tempCash value
 */
public void setTempCash(float tempCash) {
		this.tempCash = tempCash;
}
/*
 * Getter method to return cash value
 */
public float getCash() {
		return cash;
}
/*
 * Setter method to store cash value
 */
	public void setCash(float cash) {
		this.cash = cash;
	}
	/*
	 * Getter method to return PriceI value
	 */
	public int getPriceI() {
		return priceI;
	}
	/*
	 * Setter method to store priceI value
	 */
	public void setPriceI(int priceI) {
		this.priceI = priceI;
	}
	/*
	 * Getter method to return liter value
	 */
	public int getLiter() {
		return liter;
	}
	/*
	 * Setter method to store liter value
	 */
	public void setLiter(int liter) {
		this.liter = liter;
	}
	/*
	 * Getter method to return totalI value
	 */
	public int getTotalI() {
		return totalI;
	}
	/*
	 * Setter method to store totalI value
	 */
	public void setTotalI(int totalI) {
		this.totalI = totalI;
	}

}
```

11.StateContext.java

```
package state;

import abstractfactory.AbstractFactory;
import strategy.OP;

/**
 * This class represents the context class which manages the states by a
 * currState pointer as a reference to forward calls to specific state classes.
 *
 * @author cheth
 *
 */
public class StateContext {

	private State currState;
	public OP op;

	/*
	 * Constructor to initialize op object and point currState to Start state.
	 */
	public StateContext(AbstractFactory factory) {
		op = new OP(factory);
		currState = new Start();
	}
```

```
/*
 * Change current state according to the state passed as argument
 */
public void setCurrent(State s){
		currState = s;
}

/*
 * Call activate() of current state pointing to.
 */
public void activate(){
		currState.activate(this);
}

/*
 * Function to call start() of current state pointing to.
 */
public void start(){
		currState.start(this);
}

/*
 * Call payType() of current state pointing to.
 */
public void payType(int t){
		currState.payType(t,this);
}

/*
 * Call approved() of current state pointing to.
 */
public void approved(){
		currState.approved(this);
}

/*
 * Call reject() of current state pointing to
 */
public void reject(){
		currState.reject(this);
}

/*
 * Call selectGas() of current state pointing to
 */
public void selectGas(int g){
		currState.selectGas(g,this);
}

/*
 * Call cancel() of current state pointing to
 */
public void cancel(){
		currState.cancel(this);
}

/*
 * Call startPump() of current state pointing to
 */
public void startPump(){
		currState.startPump(this);
}

/*
 * Call pump () of current state pointing to.
 */
public void pump(){
		currState.pump(this);
	}

	/*
	 * Call stopPump() of current state pointing to.
	 */
	public void stopPump(){
		currState.stopPump(this);
	}

	/*
	 * Call receipt() of current state pointing to.
	 */
	public void receipt(){
		currState.receipt(this);
	}

	/*
	 * Call noReceipt() of current state pointing to.
	 */
	public void noReceipt(){
		currState.noReceipt(this);
	}
}
```

12. State.java

```
package state;
/**
 * This class represents the abstract class for States.
 * Child state classes give the concrete implementation for all the functions in the current class.
 * @author cheth
 *
 */
public abstract class State {

	public void activate(StateContext sm){}
	public void start(StateContext sm){}
	public void payType(int t,StateContext sm){}
	public void approved(StateContext sm){}
	public void reject(StateContext sm){}
	public void selectGas(int g,StateContext sm){}
	public void cancel(StateContext sm){}
	public void startPump(StateContext sm){}
	public void pump(StateContext sm){}
	public void stopPump(StateContext sm){}
	public void receipt(StateContext sm){}
	public void noReceipt(StateContext sm){}

	public void insertBorder(){
		System.out.println("---------------------");
	}
}
```

13. Start.java

```
package state;

/**
 * This class represents the concrete class for Start state.
 * @author cheth
 *
 */
public class Start extends State{

	public Start() {
		insertBorder();
		System.out.println("Entering start state..");
		insertBorder();
	}

	/*
	 * Gas pump is activated by storing the data(prices of gas types) through OP class and sets the current state 
pointer to S0.
	 * @see state.State#activate(state.StateContext)
	 */
	public void activate(StateContext sm){
		sm.op.storeData();
		sm.setCurrent(new S0());
	}
}
```

14. S0.java

```
package state;

/*
 * This class represents the concrete class for S0 state.
 */
public class S0 extends State{

	public S0() {
		insertBorder();
		System.out.println("Entering S0 State..");
		insertBorder();
	}

	/*
	 * GasPump1: Pay Credit is displayed through OP class and sets the current state pointer to S1.
	 * GasPump2: Pay cash is displayed through OP class and sets the current state pointer to S1.
	 * @see state.State#start(state.StateContext)
	 */
	public void start(StateContext sm){
		sm.op.payMsg();
		sm.setCurrent(new S1());
	}
}
```

15. S1.java

```
package state;
/**
 * This class represents the concrete class for S1 state.
 * @author cheth
 *
 */
public class S1 extends State{
	public S1() {
		insertBorder();
		System.out.println("Entering S1 State..");
		insertBorder();
	}

	/*
	 * GasPump1: T argument has a value of 1 which suggests credit payment and sets the current state pointer to 
S2.
	 * GasPump2: T argument has a value of 2 which suggests cash payment stores the cash and display menu 
through OP class and sets the current state pointer to S3.
	 * @see state.State#payType(int, state.StateContext)
	 */
	public void payType(int t,StateContext sm){
		if(t == 1){
			sm.setCurrent(new S2());
		} else if (t == 2){
			sm.op.storeCash();
			sm.op.displayMenu();
```

```
                        sm.setCurrent(new S3());
                }
        }
}
```

16.S2.java

```
package state;
/**
 * This class represents the concrete class for S2 state.
 * @author cheth
 *
 */
public class S2 extends State{

        public S2() {
                insertBorder();
                System.out.println("Entering S2 State..");
                insertBorder();
        }

        /*
         * GasPump1: Payment approved message is shown through OP class and sets the current state pointer to S3.
         * @see state.State#approved(state.StateContext)
         */
        public void approved(StateContext sm){
                sm.op.displayMenu();
                sm.setCurrent(new S3());
        }

        /*
         * GasPump1: Payment rejected message is shown through OP class and sets the current state pointer to S0.
         * @see state.State#reject(state.StateContext)
         */
        public void reject(StateContext sm){
                sm.op.rejectMsg();
                sm.setCurrent(new S0());
        }
}
```

17.S3.java

```
package state;

/**
 * This class represents the concrete class for S3 state.
 * @author cheth
 *
 */
public class S3 extends State{

        public S3() {
                insertBorder();
                System.out.println("Entering S3 state..");
                insertBorder();
        }
        /*
         * Transaction Cancelled message is shown through OP class and current state pointer is sets the to S0.
         * @see state.State#cancel(state.StateContext)
         */
        public void cancel(StateContext sm){
                sm.op.cancelMsg();
                sm.op.returnCash();
                sm.setCurrent(new S0());
        }
        /*
         * Gas price is stored based on the gas type selected and current state pointers is set to S4.
```

```
	 * @see state.State#selectGas(int, state.StateContext)
	 */
	public void selectGas(int g,StateContext sm){
		sm.op.setPrice(g);
		sm.setCurrent(new S4());
	}
}
```

18.S4.java

```
package state;

/**
 * This class represents the concrete class for S4 state.
 * @author cheth
 *
 */
public class S4 extends State{

	public S4() {
		insertBorder();
		System.out.println("Entering S4 state..");
		insertBorder();
	}
	/*
	 * Gas pump is started by initializing gas pump parameters and “Ready to pump” message is shown through
OP class and current state pointer is set to S5.
	 * @see state.State#startPump(state.StateContext)
	 */
	public void startPump(StateContext sm){
		sm.op.setInitialValues();
		sm.op.readyMsg();
		sm.setCurrent(new S5());
	}
}
```

19.S5.java

```
package state;

/**
 * This class represents the concrete class for S5 state.
 * @author cheth
 *
 */
public class S5 extends State{

	public S5() {
		insertBorder();
		System.out.println("Entering S5 state..");
		insertBorder();
	}

	/*
	 * A liter or gallon is pumped and gas units pumped is shown through OP class.
	 * @see state.State#pump(state.StateContext)
	 */
	public void pump(StateContext sm){
		sm.op.pumpGasUnit();
		sm.op.gasPumpedMsg();
	}
	/*
	 * Gas pump is stopped message is displayed and current state pointer is pointing to S6.
	 * @see state.State#stopPump(state.StateContext)
	 */
	public void stopPump(StateContext sm){
		sm.op.stopMsg();
```

```
                sm.setCurrent(new S6());
        }
}
```

20.S6.java

```
package state;

/**
 * This class represents the concrete class for S6 state.
 * @author cheth
 *
 */
public class S6 extends State{

        public S6() {
                insertBorder();
                System.out.println("Entering S6 state..");
                insertBorder();
        }

        /*
         * Receipt is generated for the gas disposed and cash left is returned depending on the gas pump through OP
class and current state pointer is set to S0.
         * @see state.State#receipt(state.StateContext)
         */
        public void receipt(StateContext sm){
                sm.op.printReceipt();
                sm.op.returnCash();
                sm.setCurrent(new S0());
        }

        /*
         * Only cash left is returned for gas pump 2 through OP class and current state pointer is set to S0.
         * @see state.State#noReceipt(state.StateContext)
         */
        public void noReceipt(StateContext sm){
                sm.op.returnCash();
                sm.setCurrent(new S0());
        }
}
```

21.OP.java

```
package strategy;

import abstractfactory.AbstractFactory;
import datastore.DataStore;

/**
 * This class represents the output processor for MDA and is the client of actions for various strategies.
 * This contains multiple interfaces which are implemented by classes devising strategies.
 * @author Chethan
 *
 */
public class OP {

        private AbstractFactory abstractFactory;
        private DataStore dataStore;

        private A1 a1;
        private A2 a2;
        private A3 a3;
        private A4 a4;
        private A5 a5;
        private A6 a6;
        private A7 a7;
        private A8 a8;
        private A9 a9;
        private A10 a10;
        private A11 a11;
        private A12 a12;
        private A13 a13;
        private A14 a14;

        public OP(AbstractFactory factory) {
                abstractFactory = factory;
                dataStore = abstractFactory.getDataStore();

                a1 = abstractFactory.getA1();
                a2 = abstractFactory.getA2();
                a3 = abstractFactory.getA3();
                a4 = abstractFactory.getA4();
                a5 = abstractFactory.getA5();
                a6 = abstractFactory.getA6();
                a7 = abstractFactory.getA7();
                a8 = abstractFactory.getA8();
                a9 = abstractFactory.getA9();
                a10 = abstractFactory.getA10();
                a11 = abstractFactory.getA11();
                a12 = abstractFactory.getA12();
                a13 = abstractFactory.getA13();
                a14 = abstractFactory.getA14();
        }

        public void storeData(){
                a1.storeData(dataStore);
        }

        public void payMsg(){
                a2.payMessage();
        }

        public void storeCash(){
                a5.storeCash(dataStore);
        }

        public void displayMenu(){
                a6.displayMenu();
        }

        public void rejectMsg(){
                a3.rejectMessage();
        }

        public void setPrice(int g){
                a7.setPrice(g);
        }

        public void readyMsg(){
                a9.readyMsg();
        }

        public void setInitialValues(){
                a8.setInitialValues();
        }

        public void pumpGasUnit(){
                a10.pumpGasUnit();
        }

        public void gasPumpedMsg(){
                a11.gasPumpedMsg();
        }
```

```
		public void stopMsg(){
			a12.stopMsg();
		}

		public void printReceipt(){
			a13.printReceipt();
		}

		public void cancelMsg(){
			a4.cancelMessage();
		}

		public void returnCash(){
			a14.returnCash();
		}
}
```

22. A1

```
package strategy;
import datastore.DataStore;

/**
 * This class represents the interface for various strategies of storeData() operation.
 * @author cheth
 *
 */
public interface A1 {

		public void storeData(DataStore dataStore);
}
```

23. A1_a.java

```
package strategy;
import datastore.DataStore;
/**
 * This class implements A1 interface by overriding the abstract method with a specific strategy for GasPump1.
 * @author cheth
 *
 */
public class A1_a implements A1{

		DataStore dataStore;
		public A1_a(DataStore ds) {
			dataStore = ds;
		}
		/*
		 * Set Regular and Super gas price from the temporary variables.
		 */
		public void storeData(DataStore dataStore) {
			dataStore.setRPriceF(dataStore.getTempRPriceF());
			dataStore.setSPriceF(dataStore.getTempSPriceF());
		}


}
```

24. A1_b.java

```
package strategy;
import datastore.DataStore;
/**
 * This class implements A1 interface by overriding the abstract method with a specific strategy for GasPump2.
 * @author cheth
 *
 */
public class A1_b implements A1{
	DataStore dataStore;
	public A1_b(DataStore ds) {
		dataStore = ds;
	}

	/*
	 * Set Regular and Super gas price from the temporary variables.
	 */
	public void storeData(DataStore dataStore) {
		dataStore.setRPriceI(dataStore.getTempRPriceI());
		dataStore.setSPriceI(dataStore.getTempSPriceI());
		dataStore.setPPriceI(dataStore.getTempPPriceI());
	}


}
```

25.A2.java

```
package strategy;

/**
 * This class represents the interface for various strategies of payMessage() operation.
 * @author cheth
 *
 */
public interface A2 {

	public void payMessage();
}
```

26.A2_a.java

```
package strategy;

public class A12_a implements A12{

	public void stopMsg() {
		System.out.println("Gas Pump Stopped..\n");
	}
}
```

27.A2_b.java

```
package strategy;
/**
 * This class implements A2 interface by overriding the abstract method with a specific strategy for GasPump2.
 * @author cheth
 *
 */
public class A2_b implements A2{
	/*
	 * Displays “Pay Cash” message.
	 */
	public void payMessage() {
		System.out.println("\nPayment mode: Cash\n");
	}
}
```

28.A3.java

```
	package strategy;
/**
 * This class represents the interface for various strategies of rejectMessage() operation.
 * @author cheth
 *
 */
public interface A3 {

        public void rejectMessage();
}
```

29.A3_a.java

```
package strategy;

/**
 * This class implements A3 interface by overriding the abstract method with a generic strategy for both the GasPumps.
 * @author cheth
 *
 */
public class A3_a implements A3 {
        /*
         * Displays "Payment Rejected" message
         */
        public void rejectMessage() {
                System.out.println("Payment Rejected!\n");
        }
}
```

30.A4.java

```
package strategy;
/**
 * This class implements A4 interface by overriding the abstract method with a generic strategy for both the GasPumps.
 * @author cheth
 *
 */
public interface A4 {

        public void cancelMessage();
}
```

31.A4_a.java

```
package strategy;

public class A4_a implements A4 {

        public void cancelMessage() {
                System.out.println("\nTransaction cancelled\n");
        }
}
```

32.A5.java

```
package strategy;
import datastore.*;

public interface A5 {

        public void storeCash(DataStore dataStore);
}
```

33.A5_Float.java

```
package strategy;

import datastore.*;

public class A5_Float implements A5{
	DataStore dataStore;
	public A5_Float(DataStore ds) {
		dataStore = ds;
	}

	public void storeCash(DataStore dataStore) {
		dataStore.setCash(dataStore.getTempCash());
	}

}
```

34.A6.java

```
package strategy;

public interface A6 {

	public void displayMenu();
}

package strategy;

public class A6_a implements A6{

	public void displayMenu() {
		System.out.println("\nPlease select Gas type:\n6.Regular\t7.Super\n");
	}

}
```

35.A6.java

```
package strategy;

public class A6_b implements A6{

	public void displayMenu() {
		System.out.println("\nPlease select Gas type:\n4.Premium\t5.Regular\t6.Super\n");
	}

}
```

36.A7.java

```
package strategy;

public interface A7 {

	public void setPrice(int g);
}
```

37.A7_Float.java

```
package strategy;
import datastore.DataStore;

public class A7_Float implements A7{

	DataStore dataStore;
	public A7_Float(DataStore ds){
		dataStore = ds;
	}

	public void setPrice(int g){
		if(g == 1){
			dataStore.setPriceF(dataStore.getRPriceF());
		}else if(g == 2){
			dataStore.setPriceF(dataStore.getSPriceF());
		}
	}
}
```

38. A7_Int.java

```
package strategy;
import datastore.DataStore;

public class A7_Int implements A7{

	DataStore dataStore;
	public A7_Int(DataStore ds){
		dataStore = ds;
	}

	public void setPrice(int g){
		if(g == 1){
			dataStore.setPriceI(dataStore.getRPriceI());
		}else if(g == 2){
			dataStore.setPriceI(dataStore.getSPriceI());
		}else if(g == 3){
			dataStore.setPriceI(dataStore.getPPriceI());
		}
	}
}
```

39. A8.java

```
package strategy;

public interface A8 {

	public void setInitialValues();
}

package strategy;
import datastore.DataStore;
```

40. A8.java

```
public class A8_a implements A8{

	DataStore dataStore;
	public A8_a(DataStore ds) {
		dataStore = ds;
	}

	public void setInitialValues() {
		dataStore.setGallon(0);
		dataStore.setTotalF(0);
	}

}
```

41. A8_b.java

```
package strategy;
import datastore.DataStore;

public class A8_b implements A8{

	DataStore dataStore;
	public A8_b(DataStore ds) {
		dataStore = ds;
	}

	public void setInitialValues() {
		dataStore.setLiter(0);
		dataStore.setTotalI(0);
```

```
        }

}
```

42. A9.java

```
package strategy;

public interface A9 {

        public void readyMsg();
}
```

43. A9_a.java

```
package strategy;

public class A9_a implements A9{

        public void readyMsg() {
                System.out.println("\nReady to pump gas!\n");
        }
}
```

44. A10.java

```
package strategy;

public interface A10 {

        public void pumpGasUnit();
}
```

45. A10_a.java

```
package strategy;
import datastore.DataStore;

public class A10_a implements A10{

        DataStore dataStore;
        public A10_a(DataStore ds){
                dataStore = ds;
        }

        public void pumpGasUnit() {
                //For gallons
                int gallon = dataStore.getGallon();
                gallon++;
                float total = dataStore.getPriceF() * gallon;
                dataStore.setGallon(gallon);
                dataStore.setTotalF(total);
        }
}
```

46. A10_b.java

```
package strategy;
import datastore.DataStore;

public class A10_b implements A10{

        DataStore dataStore;
        public A10_b(DataStore ds){
                dataStore = ds;
        }

        public void pumpGasUnit() {
                //For liters
                int liters = dataStore.getLiter();
                liters++;
                int total = dataStore.getPriceI() * liters;
```

```
			dataStore.setLiter(liters);
			dataStore.setTotalI(total);
		}
}
```

47. A11.java

```
package strategy;

public interface A11 {

		public void gasPumpedMsg();
}
```

48. A11_a.java

```
package strategy;

import datastore.DataStore;

public class A11_a implements A11{

		DataStore dataStore;
		public A11_a(DataStore ds) {
			dataStore = ds;
		}

		public void gasPumpedMsg() {
			System.out.println(dataStore.getGallon()+" gallons disposed");
			System.out.println("");
		}
}
```

49. A11_b.java

```
package strategy;

import datastore.DataStore;

public class A11_b implements A11{

		DataStore dataStore;
		public A11_b(DataStore ds) {
			dataStore = ds;
		}

		public void gasPumpedMsg() {
			System.out.println(dataStore.getLiter() +" liters disposed");
			System.out.println("");
		}
}
```

50. A12.java

```
package strategy;

public interface A12 {

		public void stopMsg();
}
```

51. A12_a.java

```
package strategy;

public class A12_a implements A12{

		public void stopMsg() {
			System.out.println("Gas Pump Stopped..\n");
		}
}
```

52.A13.java

```
package strategy;

public interface A13 {

	public void printReceipt();
}
```

53.A13_a.java

```
package strategy;
import datastore.DataStore;

public class A13_a implements A13 {

	DataStore dataStore;
	public A13_a(DataStore ds){
		dataStore = ds;
	}

	public void printReceipt() {
		System.out.println("\n############Receipt############\n");
		System.out.println("Total cost - "+dataStore.getTotalF());
	}
}
```

54.A13_b.java

```
package strategy;
import datastore.DataStore;

public class A13_b implements A13 {

	DataStore dataStore;
	public A13_b(DataStore ds){
		dataStore = ds;
	}

	public void printReceipt() {
		System.out.println("\n############Receipt############\n");
		System.out.println("Liters Pumped - "+dataStore.getLiter()+"ltrs");
		System.out.println("Total cost - "+dataStore.getTotalI());
	}
}
```

55.A14.java

```
package strategy;

public interface A14 {

	public void returnCash();
}
	package strategy;

	import datastore.DataStore;
	import datastore.DataStore2;

	public class A14_a implements A14 {

		DataStore dataStore;
		public A14_a(DataStore ds) {
			dataStore = ds;
		}

		public void returnCash() {
			if(dataStore instanceof DataStore2){
				float cashLeft = dataStore.getCash() -dataStore.getTotalI();
				System.out.println("\nReturn cash - "+cashLeft);
			}
		}
	}
```