REPORT

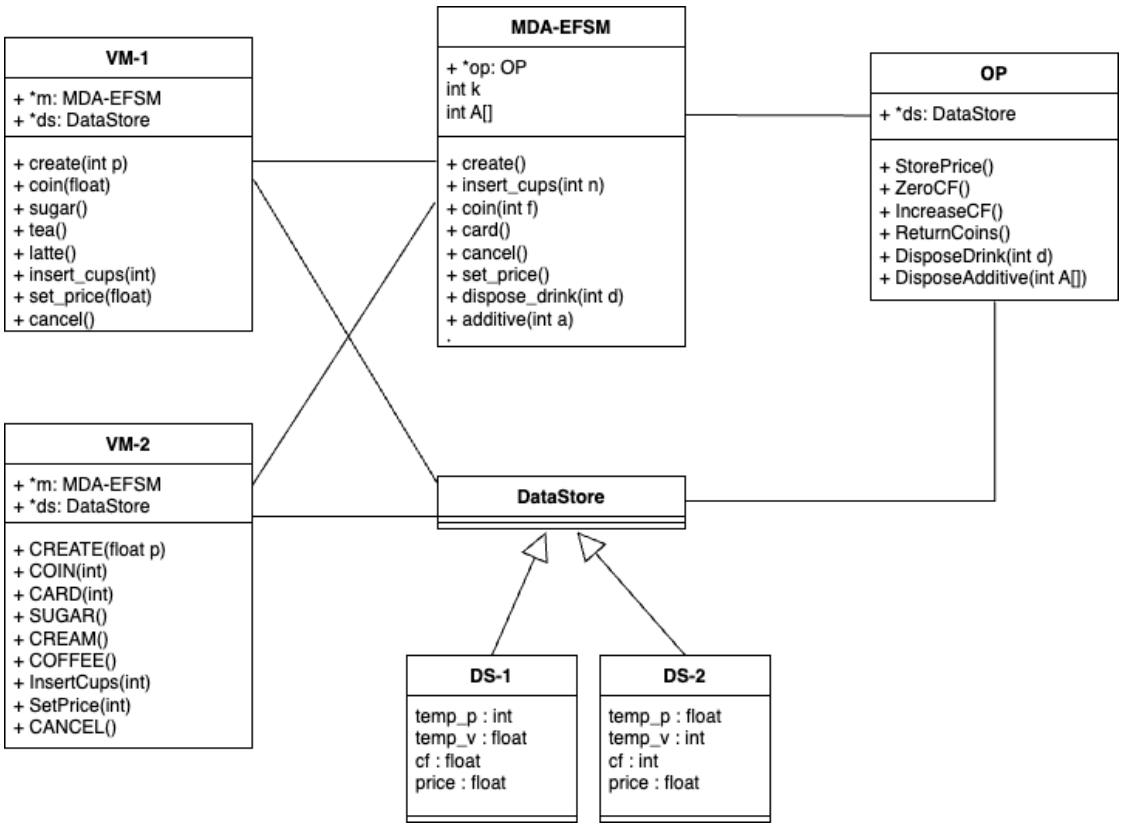
1) MDA-EFSM for Vending Machine Components

a) A list of meta events for the MDA-EFSM

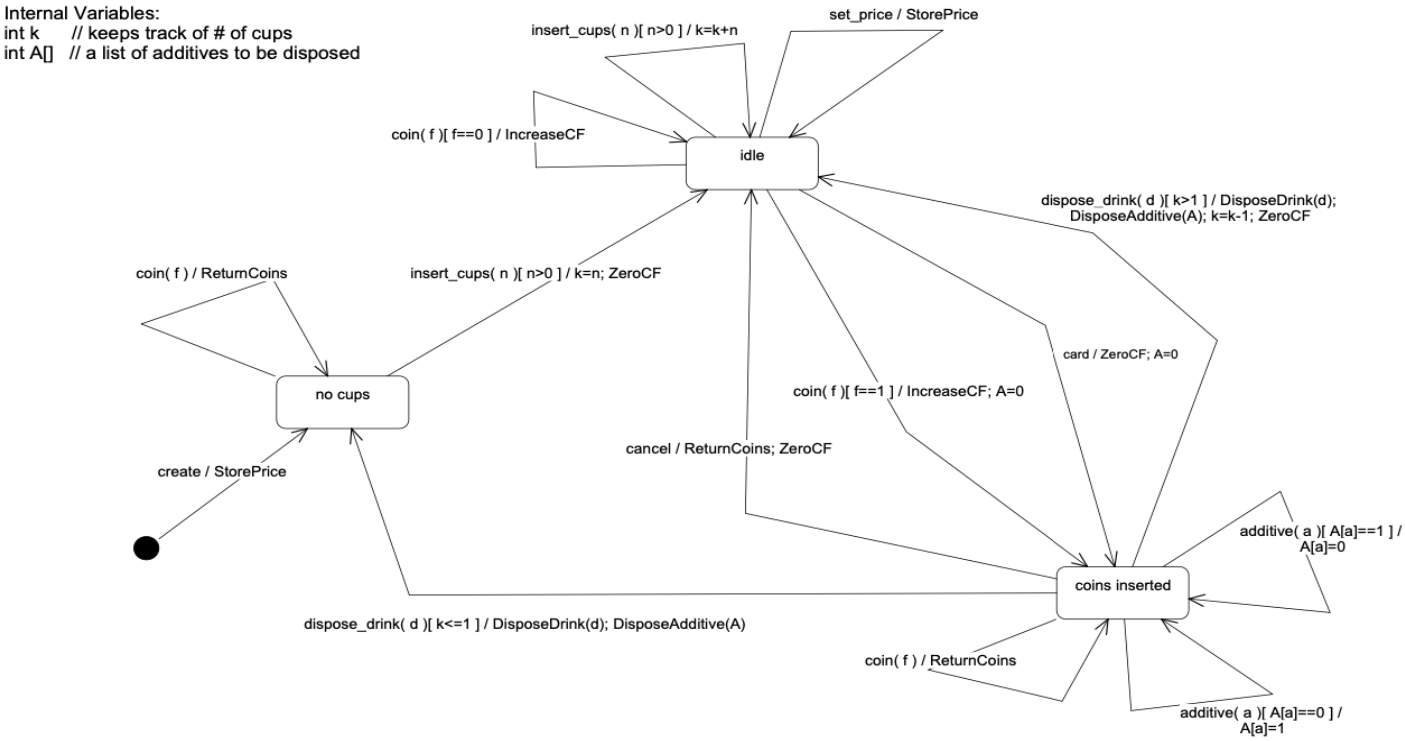
1. create()
2. insert_cups(int n) // n represents # of cups
3. coin(int f) // f=1: sufficient funds inserted for a drink.
   // f=0: not sufficient funds for a drink
4. card()
5. cancel()
6. set_price()
7. dispose_drink(int d) // d represents a drink id
8. additive(int a) // a represents additive id

b) A list of meta-actions for the MDA-EFSM with their descriptions

1. StorePrice()
2. ZeroCF() // zero Cumulative Fund cf
3. IncreaseCF() // increase Cumulative Fund cf
4. ReturnCoins() // return coins inserted for a drink
5. DisposeDrink(int d) // dispose a drink with d id
6. DisposeAdditive(int A[]) //dispose marked additives in A list,
   // where additive with i id is disposed when A[i]=1



c) A state diagram of the MDA-EFSM



d) Pseudo-code of all operations of Input Processors of Vending Machines: VM-1 and VM-2

Vending-Machine-1

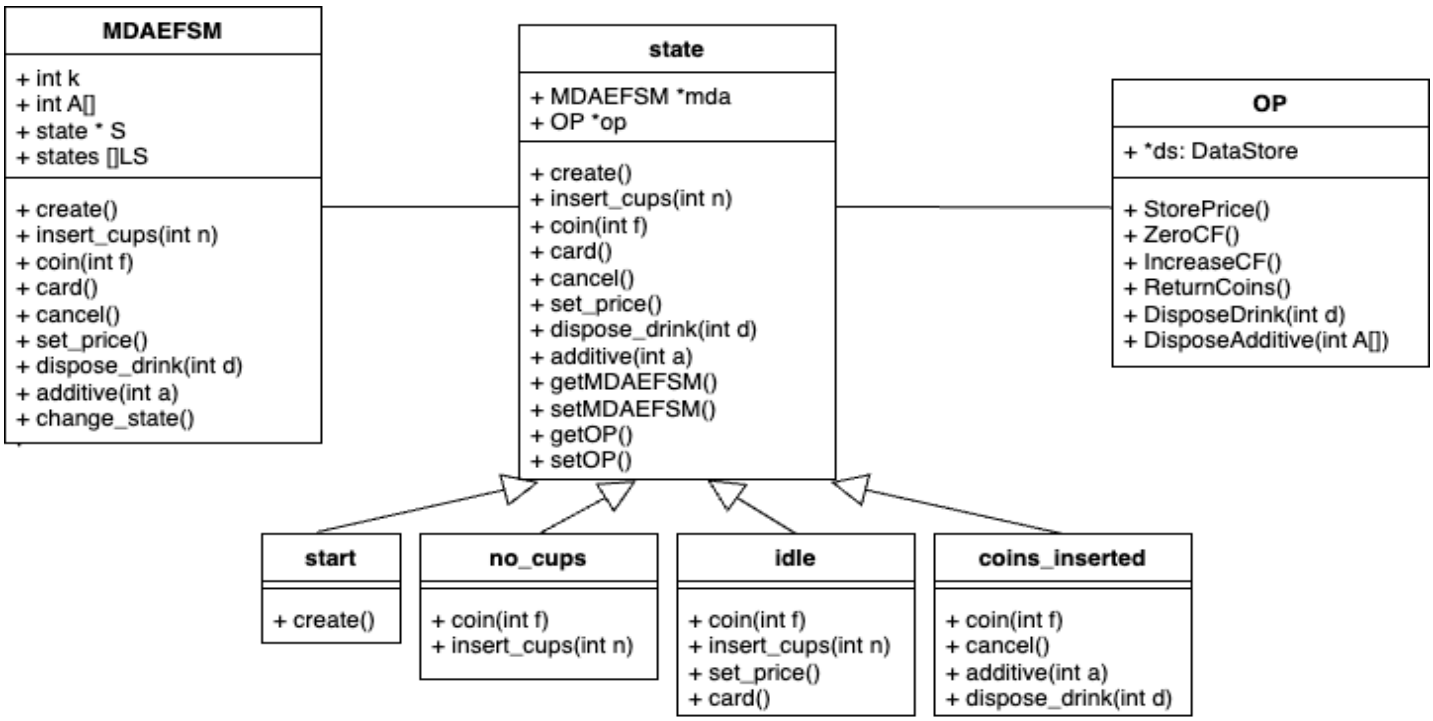
```
create(int p) {
        d->temp_p=p;
        m->create();
}

coin(float v) {
        d->temp_v=v;
        if (d->cf+v>=d->price)
                m->coin(1);
        else
                m->coin(0);
}

sugar() {
        m->additive(1);
}

tea() {
        m->dispose_drink(1);
}

latte() {
        m->dispose_drink(2);
}

insert_cups(int n) {
        m->insert_cups(n);
```

```
}

set_price(float p) {
        d->temp_p=p;
        m->set_price()
}

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

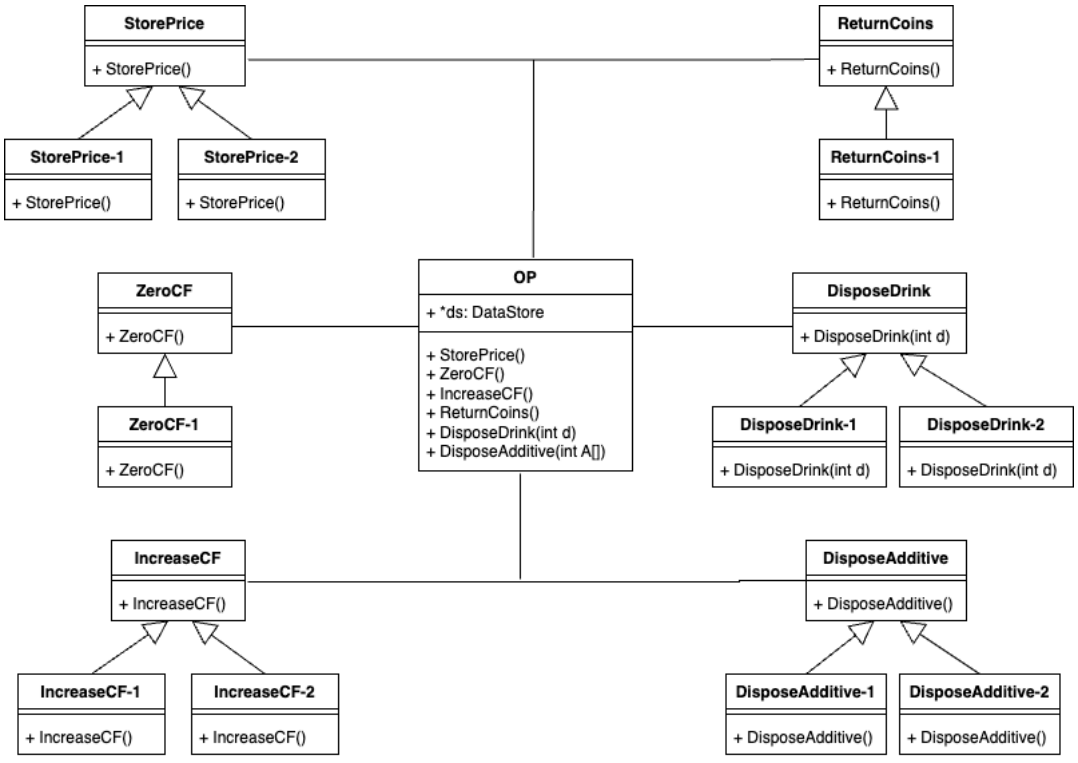
Vending-Machine-2

```
CREATE(float p) {
        d->temp_p=p;
        m->create();
}
COIN(int v) {
        d->temp_v=v;
        if (d->cf+v>=d->price)
                m->coin(1);
        else    m->coin(0);
}
CARD(int x) {
        if (x>=d->price)
                m->card();
}
SUGAR() {
        m->additive(2);
}
CREAM() {
        m->additive(1);
}
COFFEE() {
        m->dispose_drink(1);
}
InsertCups(int n) {
        m->insert_cups(n);
}
SetPrice(int p) {
        d->temp_p=p;
        m->set_price()
}
CANCEL() {
        m->cancel();
}
```

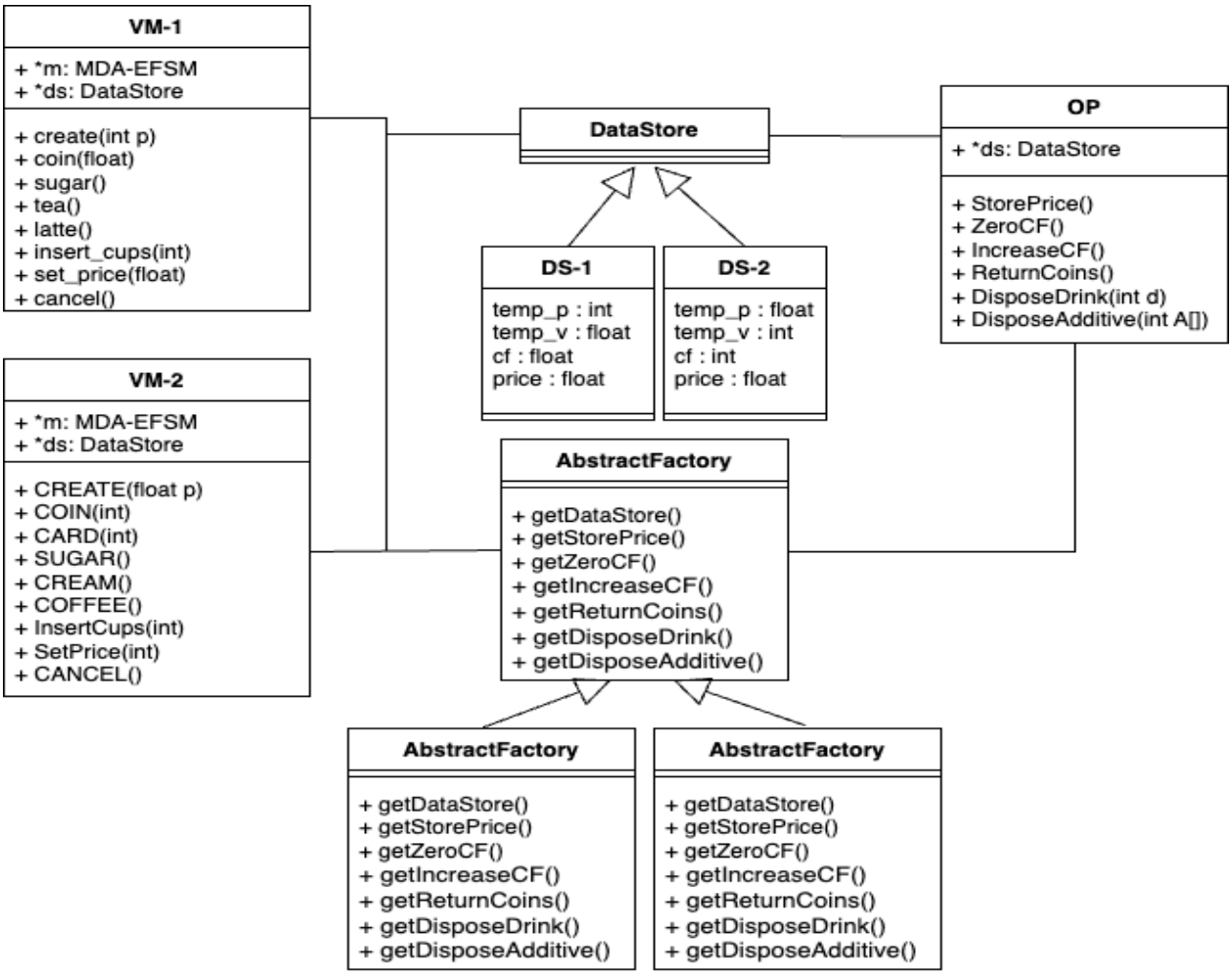
2) **Class diagram(s) of the MDA of the VM components. In your design, you MUST use the following OO design patterns:**
   a) **State pattern**

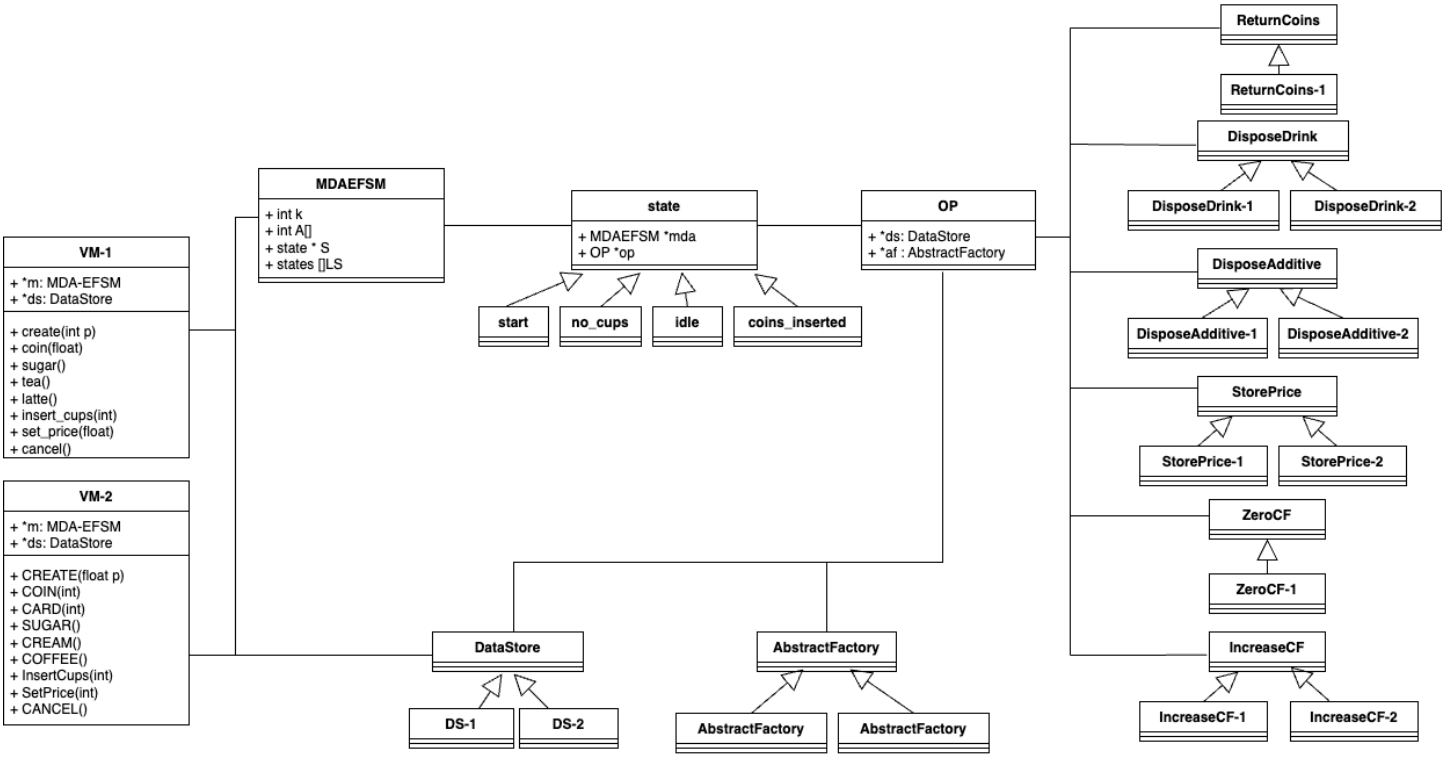


   b) **Strategy pattern**



c) Abstract factory pattern





3) For each class in the class diagram(s), you should:
   a) Describe the purpose of the class, i.e., responsibilities.
   b) Describe the responsibility of each operation supported by each class.

Class VendingMachine1 – InputProcessor

Purpose:

Defines operations of VendingMachine1

Attributes:

MDAEFSM *mda	→	pointer to MDAEFSM object
DataStore *ds	→	pointer to DataStore object
AbstractFactory *af	→	pointer to AbstractFactory class object

Methods:

create()	→	creates and stores prices of VendingMachine1
coin(float)	→	takes coins as input and performs action based on a comparision with the price of item
sugar()	→	adds sugar as an additive
tea()	→	disposes tea
latte()	→	disposes latte
insert_cups(int)	→	takes as input number of cups and inserts
set_price(float)	→	updates the pre-set price with the input price
cancel()	→	revokes any existing transaction and returns coins after coin insertion

Class VendingMachine2 – InputProcessor

Purpose:

Defines operations of VendingMachine2

Attributes:

MDAEFSM *mda	→	pointer to MDAEFSM object
DataStore *ds	→	pointer to DataStore object
AbstractFactory *af	→	pointer to AbstractFactory class object

Methods:

create()	→	creates and stores prices of VendingMachine1
coin(float)	→	takes coins as input and performs action based on a comparison with the price of item
sugar()	→	adds sugar as an additive
tea()	→	disposes tea
latte()	→	disposes latte
insert_cups(int)	→	takes as input number of cups and inserts
set_price(float)	→	updates the pre-set price with the input price
cancel()	→	revokes any existing transaction and returns coins after coin insertion

Class OutputProcessor

Purpose:

Handles and executes the actions

Attributes:

private DataStore ds	→	points to DataStore
private AbstractFactory af	→	points to AbstractFactory object
private StorePrice StorePrice	→	points to StorePrice object
private ZeroCF ZeroCF	→	points to ZeroCF object
private ReturnCoins ReturnCoins	→	points to ReturnCoins object
private IncreaseCF IncreaseCF	→	points to IncreaseCF object
private DisposeDrink DisposeDrink	→	points to DisposeDrink object
private DisposeAdditive DisposeAdditive	→	points to DisposeAdditive object

Methods:

StorePrice()	→	sets DataStore & StorePrice objects using AbstractFactory and executes StorePrice action
ZeroCF()	→	sets DataStore & StorePrice objects using AbstractFactory and executes ZeroCF action
ReturnCoins ()	→	sets DataStore & StorePrice objects using AbstractFactory and executes ZeroCF action
IncreaseCF()	→	sets DataStore & StorePrice objects using AbstractFactory and executes IncreaseCF action
DisposeDrink()	→	sets DataStore & StorePrice objects using AbstractFactory and executes DisposeDrink action
DisposeAdditive(int)	→	sets DataStore & StorePrice objects using AbstractFactory and executes DisposeAdditive action
getDataStore(float)	→	get DataStore object
setDataStore()	→	set DataStore object
getAbstractFactory()	→	get AbstractFactory object
setAbstractFactory()	→	get AbstractFactory object

Class DataStore

Purpose:

Abstract class for defining getters & setters of platform dependent parameters

Methods:

getIntTemp_p()	→ abstract getter for temporary variable int temp_p
setTemp_p()	→ abstract setter for temporary variable int temp_p
getFloatTemp_p()	→ abstract getter for temporary variable float temp_p
setTemp_p(float p)	→ abstract setter for temporary variable float temp_p
getIntTemp_v()	→ abstract getter for temporary variable int temp_v
setTemp_v(int v)	→ abstract setter for temporary variable int temp_v
getFloatTemp_v()	→ abstract getter for temporary variable float temp_v

setTemp_v(float v) → abstract setter for temporary variable float temp_v
getFloatCf() → abstract getter for temporary variable float cumulative funds
setCf(float c) → abstract setter for variable float cumulative funds
setCf(int c) → abstract getter for variable int cumulative funds
getIntCf() → abstract setter for variable int cumulative funds
getFloatPrice() → abstract getter for variable float price
setPrice(float p) → abstract setter for variable float price
getIntPrice() → abstract getter for variable int temp_p
setPrice(int p) → abstract setter for variable int temp_p

Class DataStore1

Purpose:

Abstract class for defining getters & setters of platform dependent parameters of VendingMachine1

Methods:

getIntTemp_p() → abstract getter for temporary variable int temp_p
setTemp_p() → abstract setter for temporary variable int temp_p
getFloatTemp_v() → abstract getter for temporary variable float temp_v
setTemp_v(float v) → abstract setter for temporary variable float temp_v
getFloatCf() → abstract getter for temporary variable float cumulative funds
setCf(float c) → abstract setter for variable float cumulative funds
getIntPrice() → abstract getter for variable int price
setPrice(int p) → abstract setter for variable int price

Class DataStore2

Purpose:

Abstract class for defining getters & setters of platform dependent parameters of VendingMachine2

Methods:

getFloatTemp_p() → abstract getter for temporary variable float temp_p
setTemp_p(float p) → abstract setter for temporary variable float temp_p
getIntTemp_v() → abstract getter for temporary variable int temp_v
setTemp_v(int v) → abstract setter for temporary variable int temp_v
setCf(int c) → abstract getter for variable int cumulative funds
getIntCf() → abstract setter for variable int cumulative funds
getFloatPrice() → abstract getter for variable float price
setPrice(float p) → abstract setter for variable float price

Class AbstractFactory

Purpose:

Abstract class of abstract factory design. Used to create the objects of DataStore and Action classes

Methods:

getDataStore() → abstract method to create DataStore object
getStorePrice() → abstract method to create StorePrice object
getZeroCf() → abstract method to create ZeroCF object
getIncreaseCF() → abstract method to create IncreaseCF object
getReturnCoins() → abstract method to create ReturnCoins object
getDisposeDrink() → abstract method to create DisposeDrink object
getDisposeAdditive() → abstract method to create DisposeAdditive object

Class VM1Factory

Purpose:

Concrete class used to create the objects of DataStore and Action classes for VendingMachine1

Methods:

getDataStore() → abstract method to create DataStore object
getStorePrice() → abstract method to create StorePrice object
getZeroCf() → abstract method to create ZeroCF object
getIncreaseCF() → abstract method to create IncreaseCF object
getReturnCoins() → abstract method to create ReturnCoins object
getDisposeDrink() → abstract method to create DisposeDrink object
getDisposeAdditive() → abstract method to create DisposeAdditive object

Class VM2Factory

Purpose:

Concrete class used to create the objects of DataStore and Action classes for VendingMachine2

Methods:

getDataStore() → abstract method to create DataStore object
getStorePrice() → abstract method to create StorePrice object
getZeroCf() → abstract method to create ZeroCF object
getIncreaseCF() → abstract method to create IncreaseCF object
getReturnCoins() → abstract method to create ReturnCoins object
getDisposeDrink() → abstract method to create DisposeDrink object
getDisposeAdditive() → abstract method to create DisposeAdditive object

Class MDAEFSM – State Pattern

Purpose:

Handles events called by the InputProcessor for VM1 & VM2

Attributes:

State *S → points to the current State
State[] LS → list of States
Int k → stores the number of cups
Int[] A → array of additives used for disposition

Methods:

ChangeState(int) → change the current state to the state provided as parameter
create() → used to create and set price for the VM
coin(int) → used to insert coins passed as parameter
insert_cups(int) → used to insert cups passed as parameter
card() → used to set card as the mode of payment
cancel() → revoke the transaction by returning coins
set_price() → update the current price with provided price
dispose_drink(int) → dispose the drink provided in the parameter
additive(int) → dispose additive passed as parameter

Class State – State Pattern

Purpose:

Abstract class of states

Attributes:

MDAEFSM *mda → pointer to MDAEFSM object
OutputProcessor *op → pointer to OutputProcessor object

Methods:

create() → used to create and set price for the VM
coin(int) → used to insert coins passed as parameter
insert_cups(int) → used to insert cups passed as parameter
card() → used to set card as the mode of payment
cancel() → revoke the transaction by returning coins
set_price() → update the current price with provided price

dispose_drink(int) → dispose the drink provided in the parameter
additive(int) → dispose additive passed as parameter
getMDA() → getter for MDAEFSM class object
setMDA() → setter for MDAEFSM object
getOP() → getter for OutputProcessor class object
setOp() → getter for OutputProcessor class object

Class Start

Purpose:

class representing "Start" state

Methods:

create() → stores prices and changes state to "no_cups"

Class no_cups

Purpose:

class representing "no_cups" state

Methods:

coin() → updates cumulative funds with the inserted coins
insert_cups(int v) → takes number of cups as input. If v>0, sets funds to zero and updates state to "idle"

Class Idle

Purpose:

class representing "idle" state

Methods:

set_price() → calls StorePrice action to update the price
insert_cups(int v) → takes number of cups as input. If v>0, updates the number of cups i.e., k
coin(int) → if input = 1, create list of additives and update state to "coin_inserted"
card() → sets funds to 0 and updates state to "coin_inserted"

Class coins_inserted

Purpose:

class representing "coins_inserted" state

Methods:

dispose_drink (int) → dispose the drink passed as parameter based on the number of cups
additive(int v) → update the array of additives based on input parameter
coin(int) → return coins inserted
cancel() → revoke any transaction by returning coins, setting funds to 0 and changing state to "idle"

interface StorePrice

Purpose:

Strategy pattern – interface with method to execute StorePrice strategy

Methods:

StorePrice(int) → method to execute StorePrice based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds) → setter for DataStore

class StorePrice1

Purpose:

Strategy pattern – concrete implementation of StorePrice for VM1

Methods:

StorePrice(int) → stores float price for VM1

class StorePrice2

Purpose:

Strategy pattern – concrete implementation of StorePrice for VM2

Methods:

StorePrice(int) → stores float price for VM2

interface ZeroCF

Purpose:

Strategy pattern – interface with method to execute ZeroCF strategy

Methods:

ZeroCF () → method to execute ZeroCF based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds) → setter for DataStore

class ZeroCF1

Purpose:

Strategy pattern – concrete implementation of ZeroCF for VM1 & VM2

Methods:

ZeroCF (int) → sets cumulative funds to zero

interface ReturnCoins

Purpose:

Strategy pattern – interface with method to execute ReturnCoins strategy

Methods:

ReturnCoins () → method to execute ReturnCoins based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds) → setter for DataStore

class ReturnCoins

Purpose:

Strategy pattern – concrete implementation of ReturnCoins for VM1 & VM2

Methods:

ReturnCoins (int) → returns the inserted coins

interface IncreaseCF

Purpose:

Strategy pattern – interface with method to execute IncreaseCF strategy

Methods:

IncreaseCF(int) → method to execute IncreaseCF based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds) → setter for DataStore

class IncreaseCF1

Purpose:

Strategy pattern – concrete implementation of IncreaseCF for VM1

Methods:

IncreaseCF () → updated the cumulative funds for VM1

class IncreaseCF2

Purpose:

Strategy pattern – concrete implementation of IncreaseCF for VM2

Methods:

IncreaseCF () → updated the cumulative funds for VM2

interface DisposeDrink

Purpose:

Strategy pattern – interface with method to execute DisposeDrink strategy

Methods:

DisposeDrink (int) → method to execute DisposeDrink based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds) → setter for DataStore

class DisposeDrink1

Purpose:

Strategy pattern – concrete implementation of DisposeDrink for VM1

Methods:

DisposeDrink () → disposes drink based on the given input for VM1

class DisposeDrink2

Purpose:

Strategy pattern – concrete implementation of DisposeDrink for VM2

Methods:

DisposeDrink () → disposes drink based on the given input for VM2

interface DisposeAdditive

Purpose:

Strategy pattern – interface with method to execute DisposeAdditive strategy

Methods:

DisposeAdditive (int) → method to execute DisposeAdditive based on the set Strategy i.e., VM1 or VM2
getDataStore() → getter for DataStore
setdataStore(ds). → setter for DataStore

class DisposeAdditive1

Purpose:

Strategy pattern – concrete implementation of DisposeAdditive for VM1

Methods:

DisposeAdditive () → additives disposed for array values = 1 for VM1

class DisposeAdditive2

Purpose:

Strategy pattern – concrete implementation of DisposeAdditive for VM2

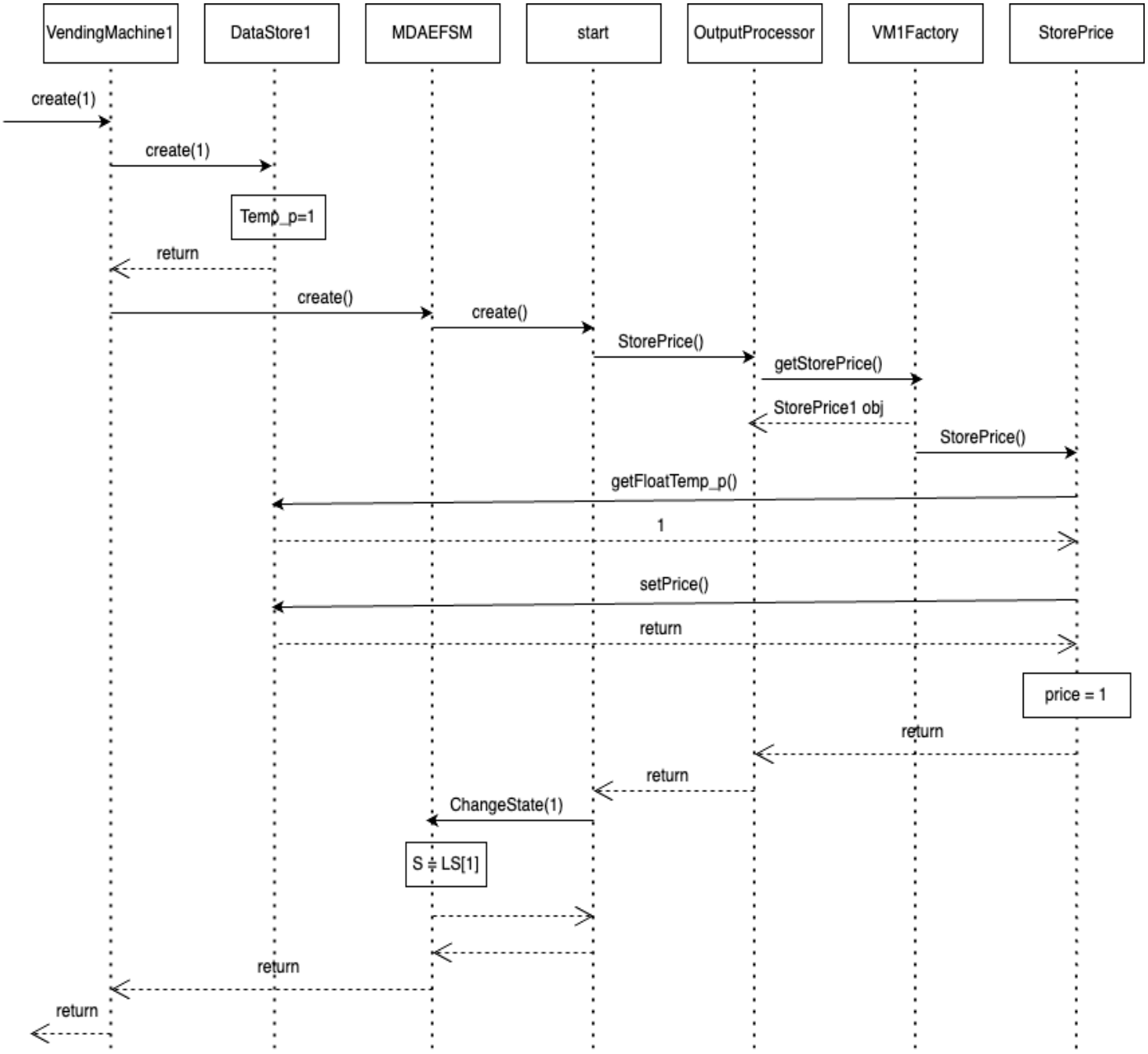
Methods:

DisposeAdditive () → additives disposed for array values = 1 for VM2

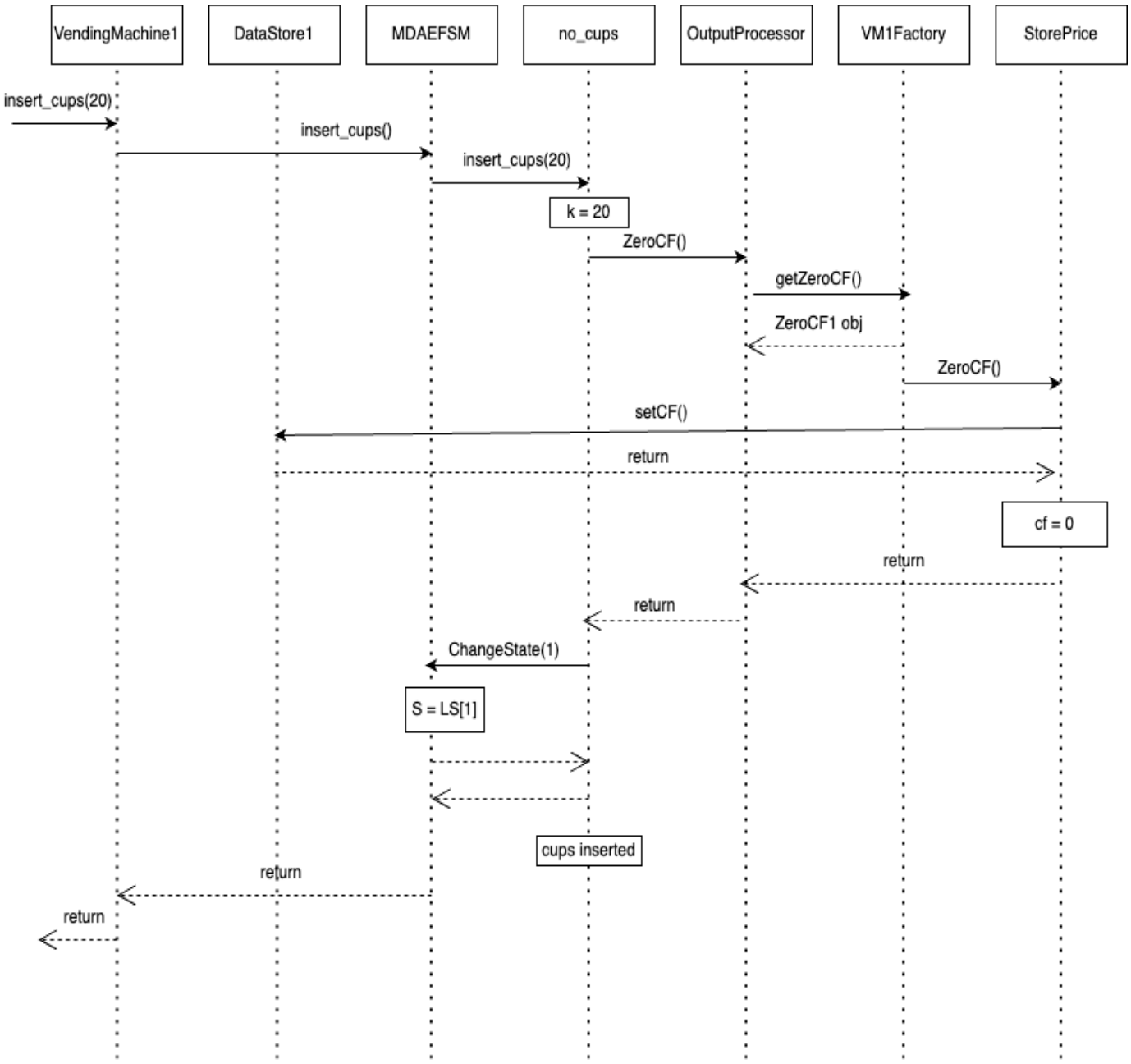
4) Dynamics. Provide two sequence diagrams for two Scenarios

Scenario-I should show how the cup of latte is disposed of in the Vending Machine VM-1 component, i.e., the following sequence of operations is issued:
create(1), insert_cups(20), coin(0.5), coin(0.5), sugar(), latte()

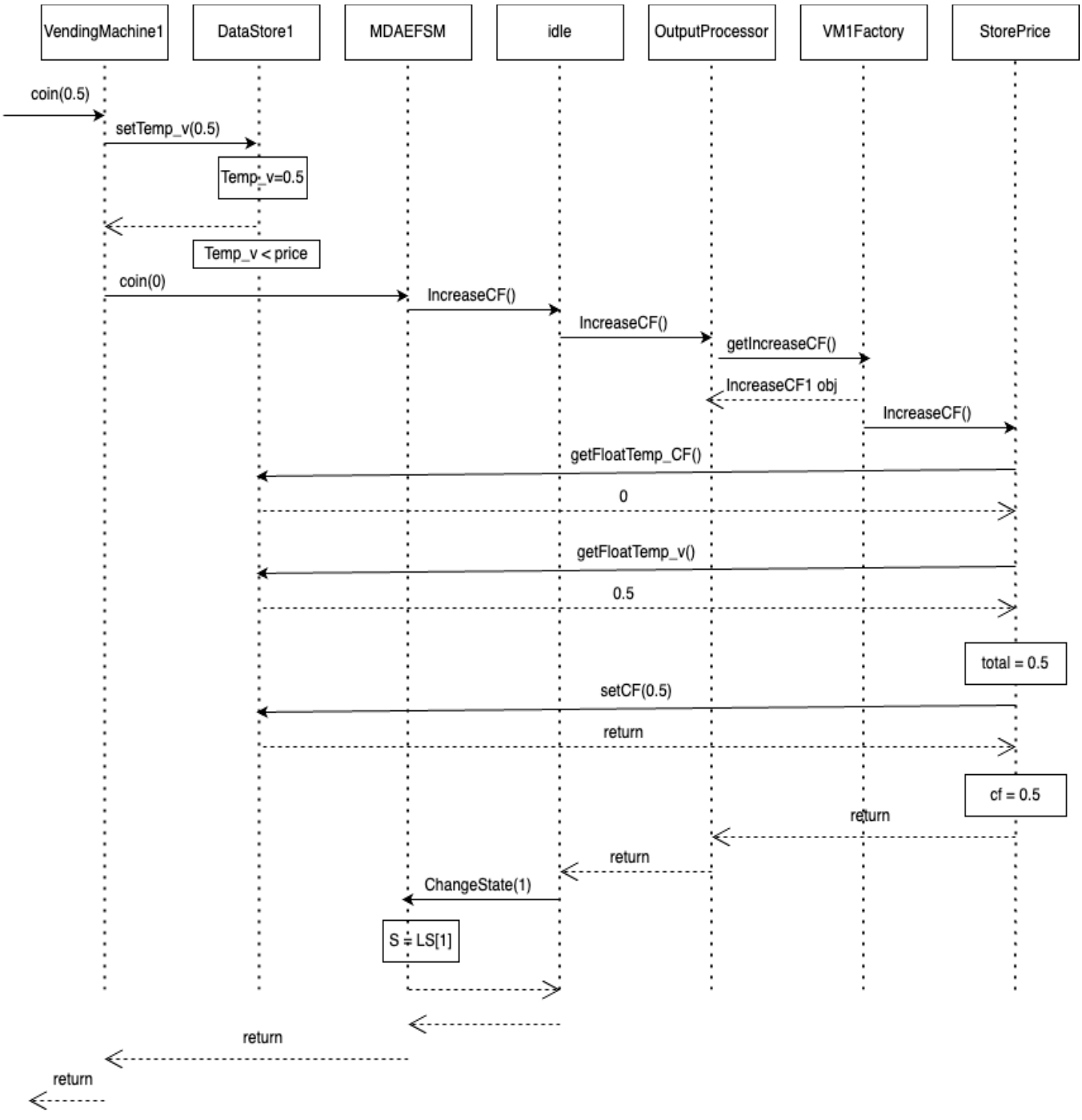
create(1)



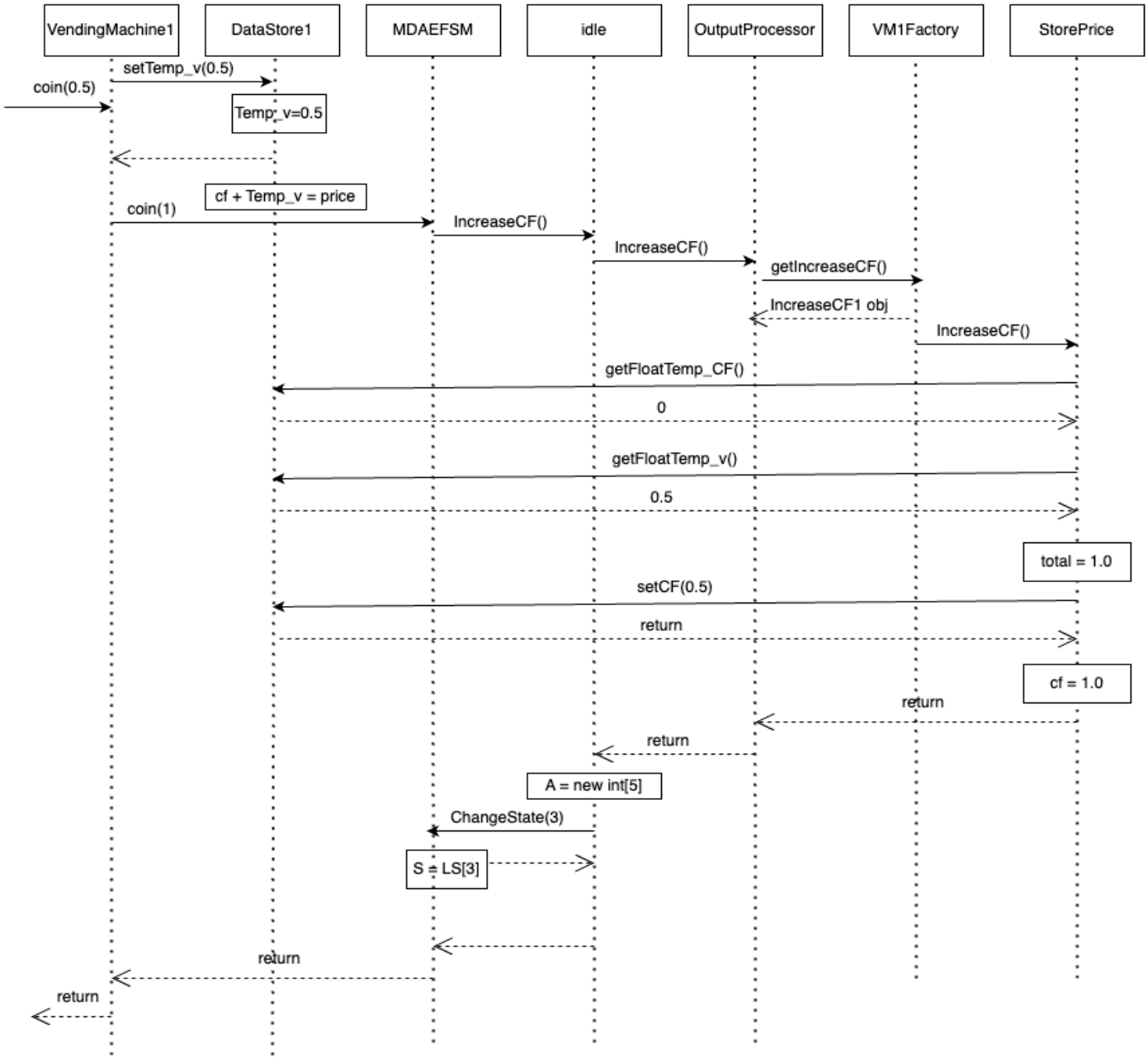
insert_cups(20)



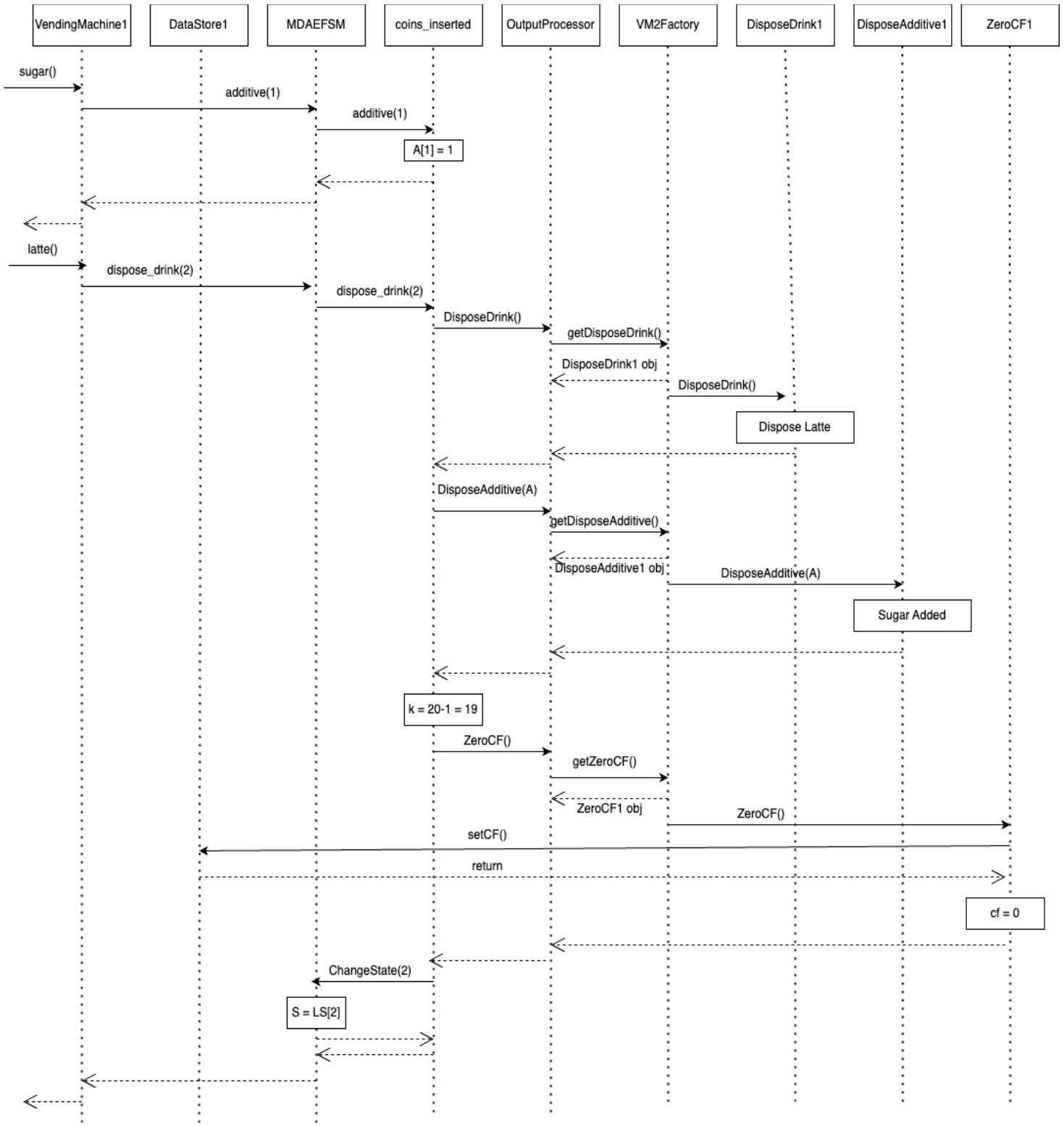
coin(0.5)


VendingMachine1
DataStore1
MDAEFSM
idle
OutputProcessor
VM1Factory
StorePrice
coin(0.5)
setTemp_v(0.5)
Temp_v=0.5
Temp_v < price
coin(0)
IncreaseCF()
IncreaseCF()
getIncreaseCF()
IncreaseCF1 obj
IncreaseCF()
getFloatTemp_CF()
0
getFloatTemp_v()
0.5
total = 0.5
setCF(0.5)
return
cf = 0.5
return
return
ChangeState(1)
S = LS[1]
return
return

coin(0.5)


VendingMachine1
DataStore1
MDAEFSM
idle
OutputProcessor
VM1Factory
StorePrice
setTemp_v(0.5)
coin(0.5)
Temp_v=0.5
cf + Temp_v = price
coin(1)
IncreaseCF()
IncreaseCF()
getIncreaseCF()
IncreaseCF1 obj
IncreaseCF()
getFloatTemp_CF()
0
getFloatTemp_v()
0.5
total = 1.0
setCF(0.5)
return
cf = 1.0
return
return
A = new int[5]
ChangeState(3)
S = LS[3]
return
return

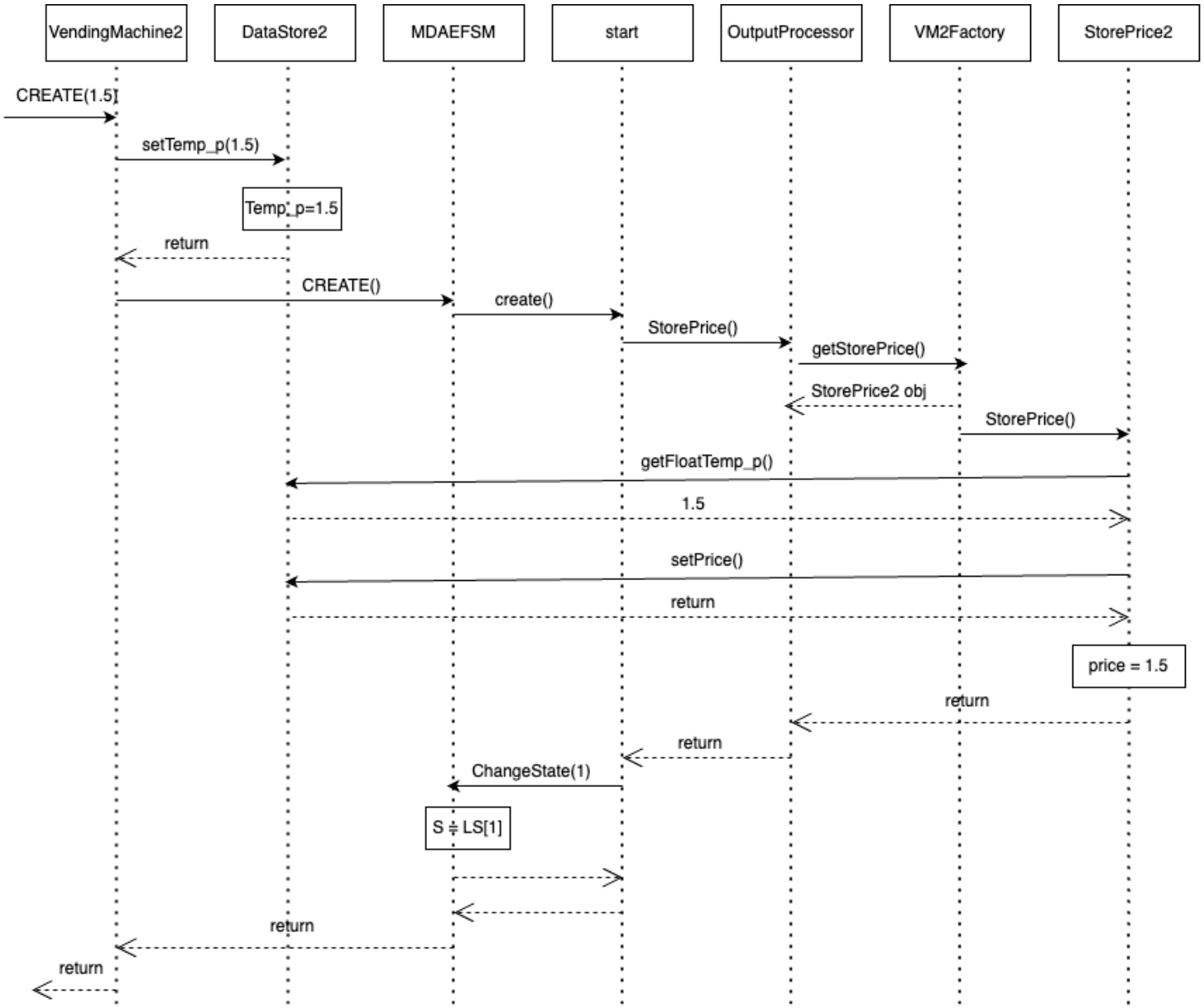
sugar(), latte()


VendingMachine1
DataStore1
MDAEFSM
coins_inserted
OutputProcessor
VM2Factory
DisposeDrink1
DisposeAdditive1
ZeroCF1
sugar()
additive(1)
additive(1)
A[1] = 1
latte()
dispose_drink(2)
dispose_drink(2)
DisposeDrink()
getDisposeDrink()
DisposeDrink1 obj
DisposeDrink()
Dispose Latte
DisposeAdditive(A)
getDisposeAdditive()
DisposeAdditive1 obj
DisposeAdditive(A)
Sugar Added
k = 20-1 = 19
ZeroCF()
getZeroCF()
ZeroCF1 obj
ZeroCF()
setCF()
return
cf = 0
ChangeState(2)
S = LS[2]

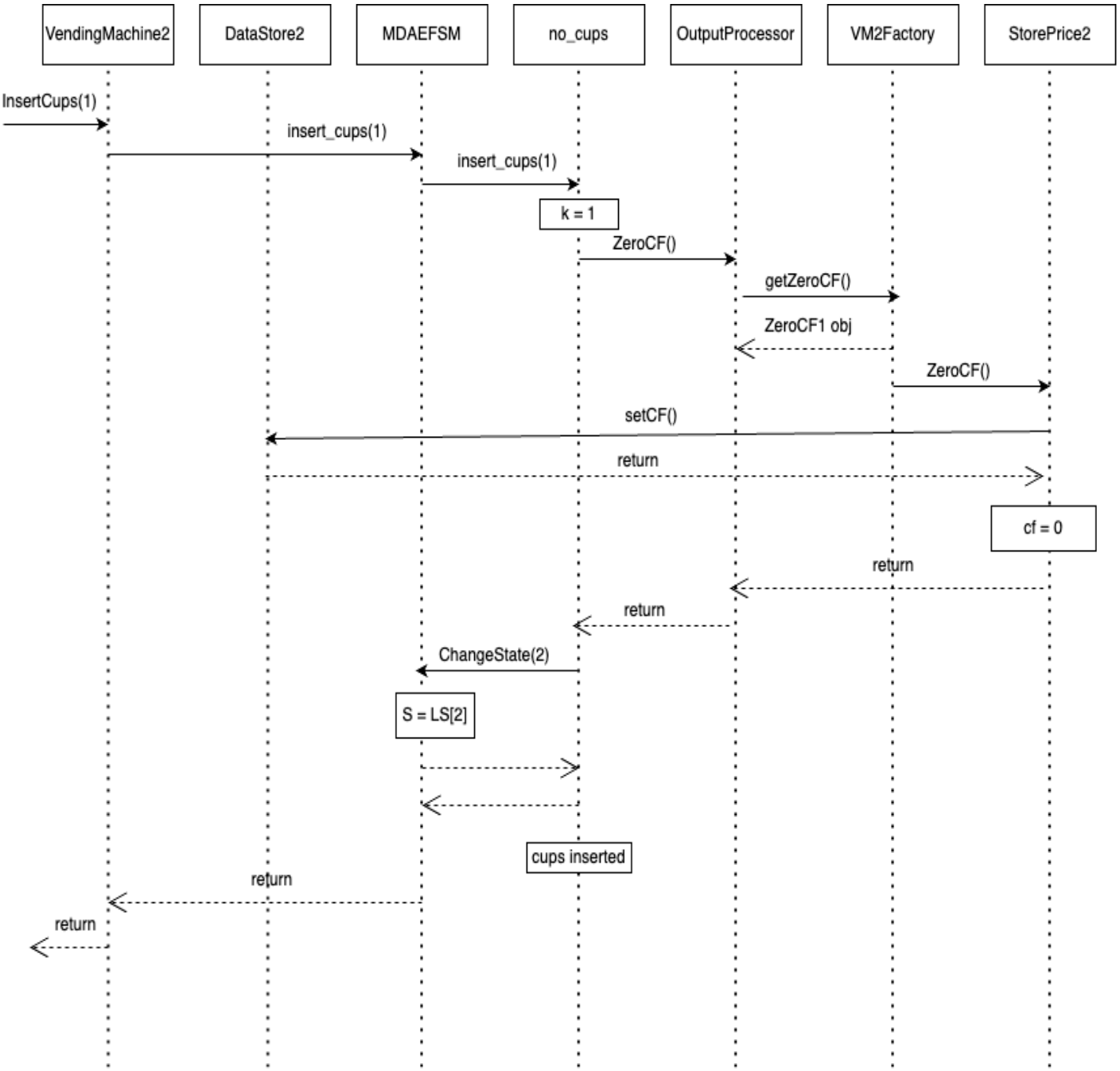
Scenario-II should show how a cup of coffee is disposed of in the Vending Machine VM-2 component, i.e., the following sequence of operations is issued:

CREATE(1.5), InsertCups(1), CARD(10), CREAM(), COFFEE()

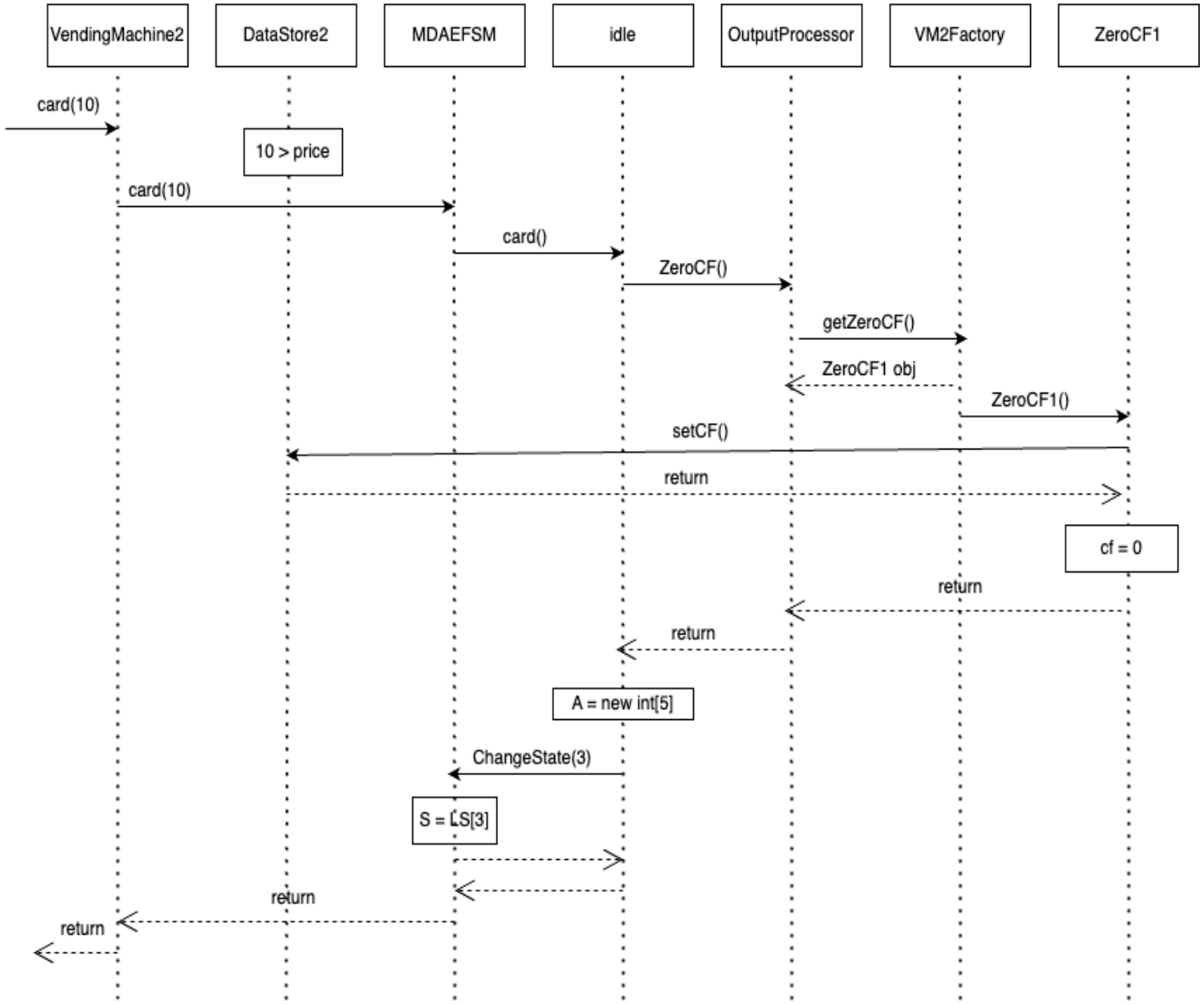
CREATE(1.5)



InsertCups(1)



CARD(10)


VendingMachine2
DataStore2
MDAEFSM
idle
OutputProcessor
VM2Factory
ZeroCF1
card(10)
10 > price
card(10)
card()
ZeroCF()
getZeroCF()
ZeroCF1 obj
ZeroCF1()
setCF()
return
cf = 0
return
return
A = new int[5]
ChangeState(3)
S = LS[3]
return
return

CREAM(), COFFEE()