SSA Project Report, A20530533, Kushagra Kasliwal

Vending Machine System REPORT

1) MDA-EFSM MODEL

a. MDA-EFSM Events:

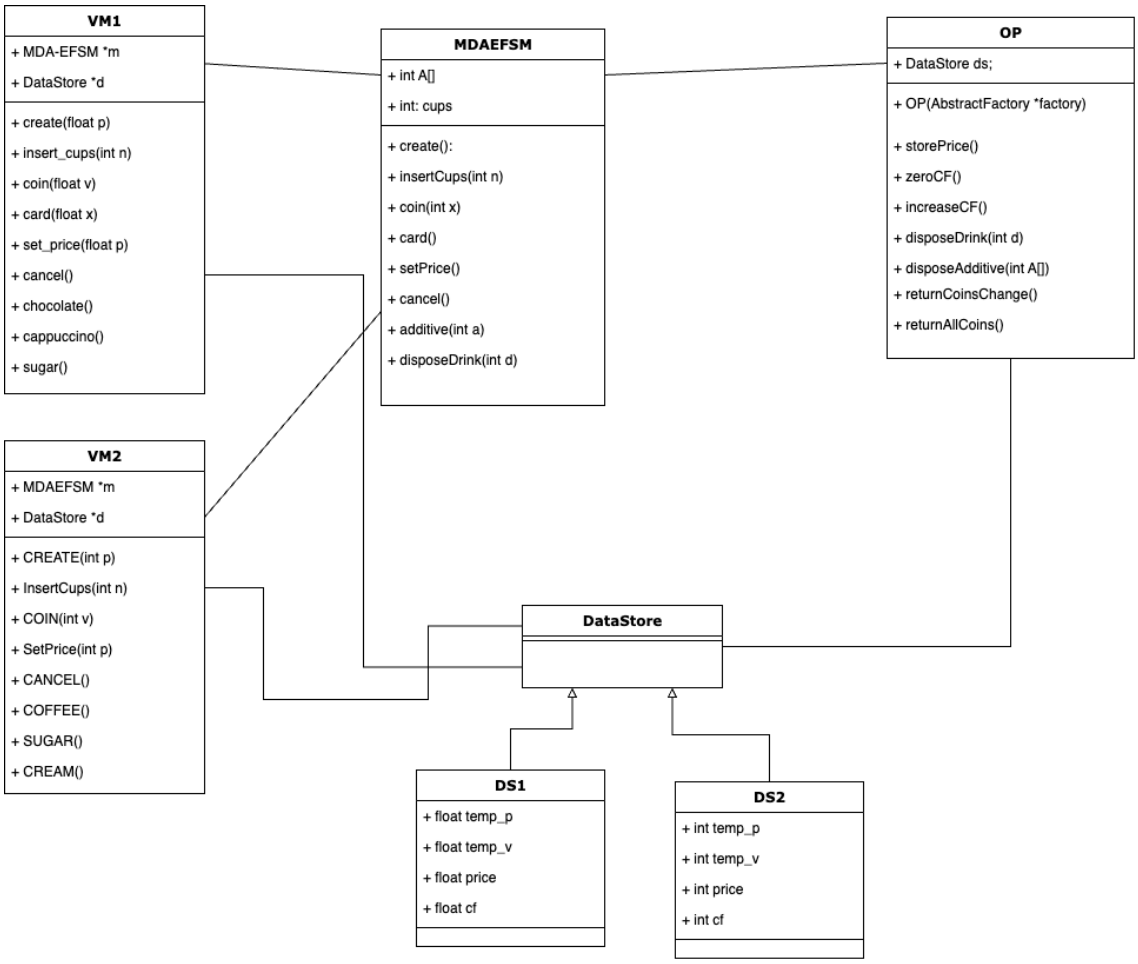
1. **create**()
2. **insertCups(int n)** // n represents # of cups
3. **coin(int f)** // f=1: sufficient funds inserted for a drink
   // f=0: not sufficient funds for a drink
4. **card**()
5. **cancel**()
6. **setPrice**()
7. **disposeDrink(int d)** // d represents a drink id
8. **additive(int a)**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

b. MDA-EFSM Actions:

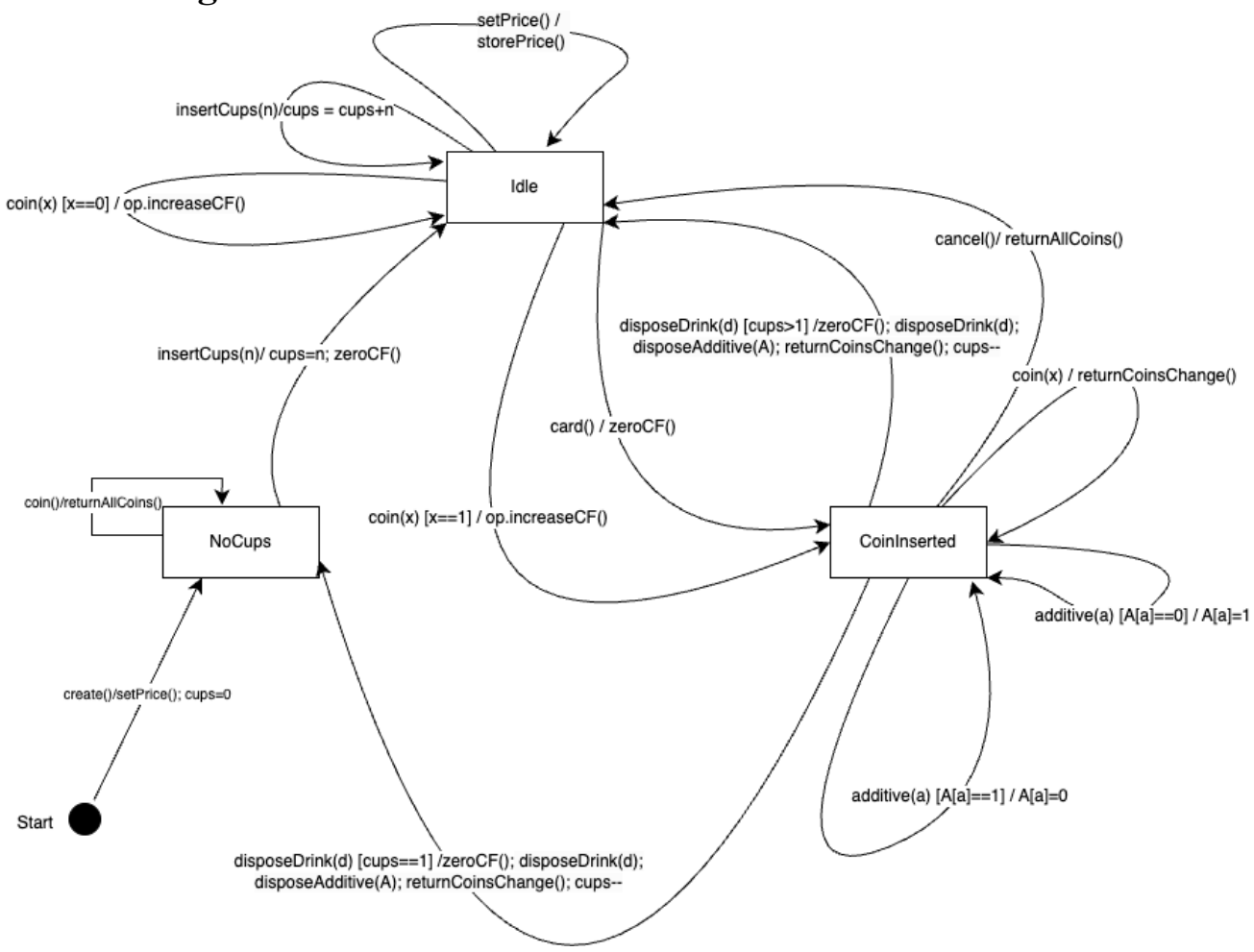
1. **storePrice**()
2. **zeroCF**() // zero Cumulative Fund cf
3. **increaseCF**() // increase Cumulative Fund cf
4. **returnAllCoins**() // returns all coins (when user cancels the operation, all coins get returned)
5. **returnCoinsChange**() //return extra coins only, (ex-> if user entered 3 coins for drink priced 2.5, it will return 0.5 coin)
6. **disposeDrink(int d)** // dispose a drink with d id
7. **disposeAdditive(int A[])** //dispose marked additives in A list,
   // where additive with i id is disposed when A[i]=1

c. Diagrams For MDA-EFSM

Class Diagram



d. State Diagram



d. Pseudo Code

VM1

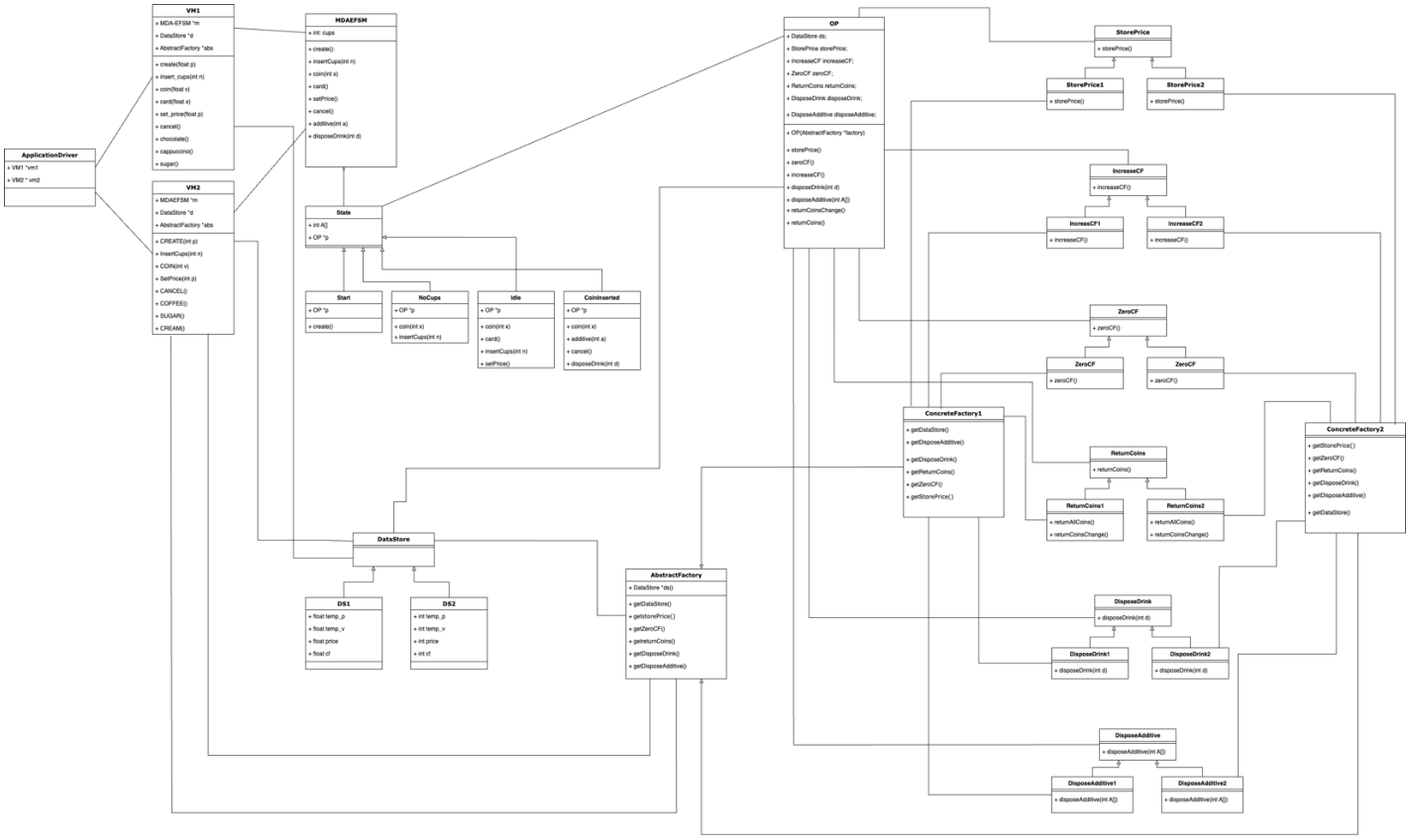
MDAEFSM m;
DataStore data;

```
create(float p) {
data->temp_p=p;
m->create();
}

insert_cups(int n) {
m->insertCups(n);
}

coin(float v) {
data->temp_v=v;
if (data->CF+v >= data->p) m->coin(1);
else m->coin(0);
}

card(float x) {
if (data->price <= x) m->card();
}

set_price(float p) {
data->temp_p =p
m.setPrice()
}

cancel() {
m.cancel()
}

cappuccino() {
m.drinkPressed(0)
}

chocolate() {
m.disposeDrink(1)
}

sugar() {
m.additive(1)
}
```

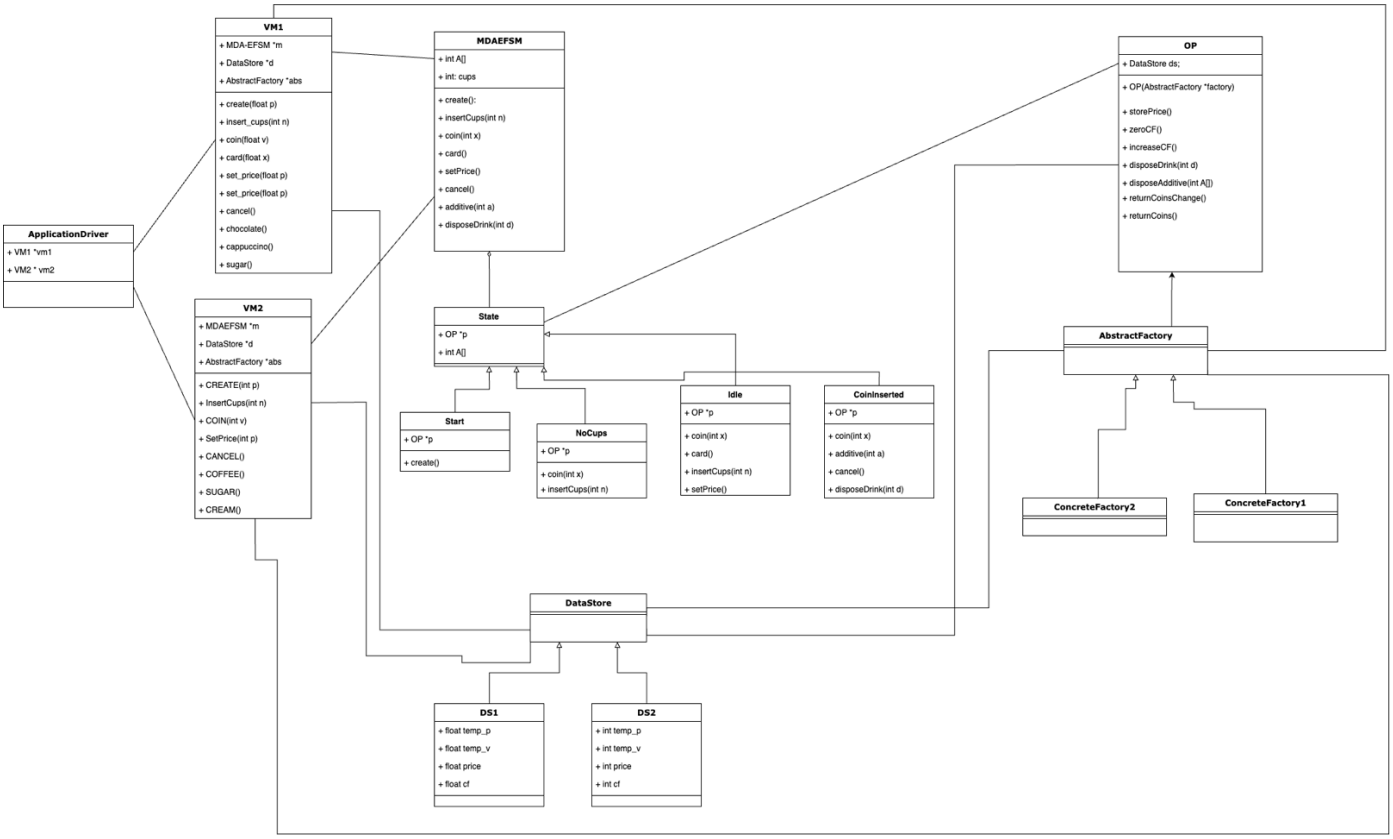
VM2

```
MDA-EFSM *m;
DataStore * data

CREATE(int p) {
data->temp_p=p;
m->create();
}

InsertCups(int n) {
if(n>0)
m->insertCups(n);
}

COIN(int v) {
data->temp_v=v;
if (data->CF+v >= data->price) m->coin(1);
else m->coin(0);
}

setPrice(int p) {
data->temp_p=p
m.setPrice()
}

cancel() {
m.cancel()
}

COFFEE() {
m.drinkPressed(0)
}

sugar() {
m.additivePressed(0)
}

cream() {
m.additivePressed(1)
}
```

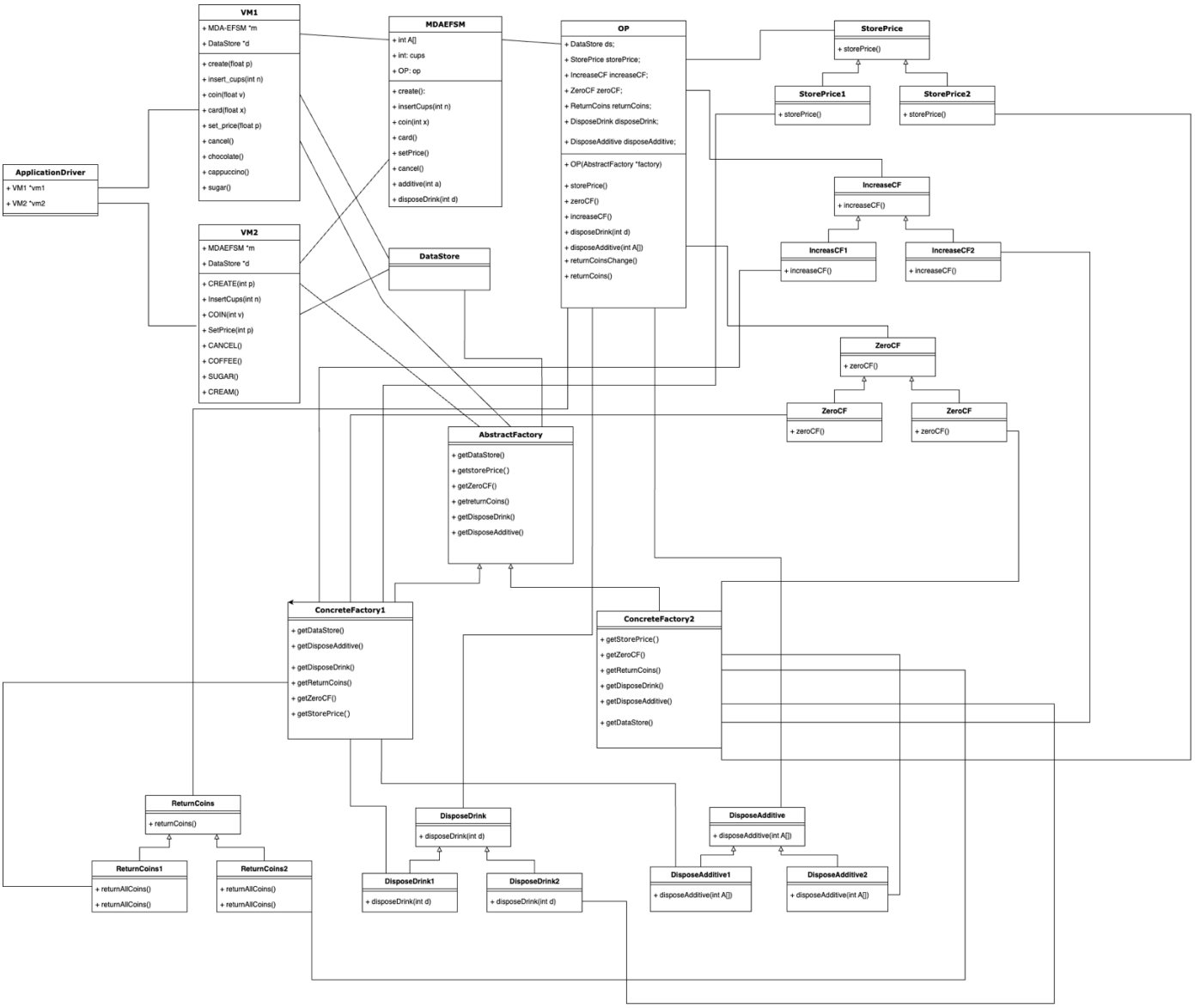
2. Class Diagrams



State Pattern



Abstract Factory and Strategy Pattern



3. Class Responsibility and operations

VM1:

Purpose:
The `VM1` class implements the user-facing interface for Vending Machine 1. It handles user operations like inserting coins, selecting drinks, or canceling a transaction. It uses an abstract factory (`ConcreteFactory1`) to access specific data and behaviors (through `DS1` and action strategies). It interacts with `MDAEFSM` to trigger state transitions based on user input.

Member Variables:

1. **`mdaEfsm`** – Manages the state transitions of the vending machine.
2. **`factory`** – Provides appropriate data store and operation strategy objects for VM1.
3. **`dataStore`** – Stores temporary and permanent data specific to VM1 (like price, credit, volume).

Operations

1. **`VM1()`**
   Initializes `ConcreteFactory1`, sets up `MDAEFSM`, and retrieves the specific `DS1` data store.
2. **`create(float p)`**
   Sets the drink price temporarily and signals the state machine to create/init the machine, if the price is valid.
3. **`coin(float v)`**
   Accepts coin input, adds it to current credit, and notifies `MDAEFSM` whether credit is sufficient to proceed.
4. **`card(float x)`**
   Accepts card input and checks if balance is sufficient before proceeding to payment handling via `MDAEFSM`.

5. **`sugar()`**
   Adds sugar as an additive to the selected drink.
6. **`cappuccino()`**
   Triggers the vending of a cappuccino drink via the state machine.
7. **`chocolate()`**
   Triggers the vending of a chocolate drink via the state machine.
8. **`insert_cups(int n)`**
   Adds cups into the machine if the input number is positive.
9. **`set_price(float p)`**
   Sets a new price for the drinks after validation and signals the state machine to commit the change.
10. **`cancel()`**
    Cancels the current transaction and resets the vending state through `MDAEFSM`.

VM2 :-

Purpose:
The `VM2` class acts as the user-facing interface for Vending Machine 2. It handles the main operations such as inserting coins, selecting additives, and canceling transactions. The class uses a specific implementation of the abstract factory (`ConcreteFactory2`) to access machine-specific components such as `DS2` (the data store). All operations are forwarded to the `MDAEFSM` to handle state transitions according to the input.

Member Variables:

1. **`mdaEfsm`** – Manages the operational state transitions for VM2.
2. **`factory`** – Provides VM2-specific instances like data store and action handlers.
3. **`dataStore`** – Stores and manages all runtime data specific to VM2 (e.g., drink price, credit, coin value).

Responsibilities of Each Method:

1. **`VM2()`**
   Initializes the machine with a concrete factory (`ConcreteFactory2`), connects the state machine (`MDAEFSM`), and retrieves the appropriate data store (`DS2`).
2. **`CREATE(int p)`**
   Validates and sets the initial price of drinks. Triggers the creation process in `MDAEFSM`.

3. **`COIN(int v)`**
   Accepts a coin value, adds it to current credit, and triggers the state machine based on whether funds are sufficient to buy a drink.
4. **`SUGAR()`**
   Adds sugar to the selected drink via the state machine.
5. **`CREAM()`**
   Adds cream to the selected drink via the state machine.
6. **`COFFEE()`**
   Initiates the dispensing of coffee through the state machine.
7. **`InsertCups(int n)`**
   Refills the machine with the specified number of cups after input validation.
8. **`SetPrice(int p)`**
   Updates the drink price and commits it via the state machine.
9. **`CANCEL()`**
   Cancels the current operation and resets relevant state in the `MDAEFSM`.

MDAEFSM :-

Purpose:
The `MDAEFSM` class implements the platform-independent meta-model for the Model-Driven Architecture (MDA) system. It encapsulates the core state machine logic that governs all operations for clients like `VM1` and `VM2`. The design separates state-specific behavior into individual state classes, promoting modularity, maintainability, and ease of extension.

Member Variables:

1. `cups`: Stores the number of available cups in the vending machine.
2. `states`: An array of `State` objects representing all possible states in the EFSM (Extended Finite State Machine).
3. `currState`: Represents the current active state in the system.

Member Functions:

1. **`MDAEFSM(AbstractFactory abstractFactory)`**
   Constructor that initializes all state objects (`Start`, `NoCups`, `Idle`, and `CoinInserted`) using the given abstract factory. Sets the initial state to `Start` and initializes cup count to zero.

2. **`create()`**
   Invokes the `create()` method of the current state if the machine is in the `Start` state. Transitions to the `NoCups`state upon successful execution.
3. **`coin(int f)`**
   Handles coin insertion:
   - In `NoCups` and `CoinInserted` states: invokes `coin(f)` without changing the state.
   - In `Idle` state: invokes `coin(f)` and transitions to `CoinInserted` only if the inserted funds are sufficient (i.e., `f == 1`).
4. **`insertCups(int c)`**
   Adds `c` cups to the machine. Allowed only in the `NoCups` or `Idle` state. Transitions from `NoCups` to `Idle` upon successful execution.
5. **`card()`**
   Processes card payment. Valid only in the `Idle` state. Transitions to `CoinInserted` state after processing.
6. **`cancel()`**
   Cancels the ongoing transaction. Allowed only in the `CoinInserted` state.
7. **`setPrice()`**
   Updates the drink price. Allowed only in the `Idle` state.
8. **`disposeDrink(int d)`**
   Dispenses the selected drink. Allowed only in the `CoinInserted` state. Decrements the cup count and updates the state to `NoCups` if no cups remain, otherwise transitions to `Idle`.
9. **`additive(int a)`**
   Adds a selected additive to the drink. Allowed only in the `CoinInserted` state.

DataStore :-

Purpose:
The `DataStore` class is an abstract base class that defines a generic interface for data storage in the vending machine system. It serves as a parent to concrete subclasses such as `DS1` and `DS2`, which provide machine-specific implementations for storing temporary and persistent runtime data.

Member Variables:
None.

Member Functions:
_None directly in this class; intended to be extended and implemented by subclasses (`DS1`, `DS2`).

DS1 :-

Purpose:
The `DS1` class is a concrete implementation of the abstract `DataStore` class and is specifically designed to support the operations of VM1. It is responsible for storing and managing runtime data including the temporary price, temporary coin value, final price, and current funds (credit) inserted by the user.

Member Variables:

1. `temp_p`: Temporarily holds the price of the drink before it's confirmed.
2. `temp_v`: Temporarily stores the value of the inserted coin.
3. `price`: Holds the final confirmed price of the drink.
4. `cf`: Stores the current funds or credit entered by the user.

Member Functions:

1. `getTemp_p()` / `setTemp_p(float temp_p)`: Gets or sets the temporary price.
2. `getTemp_v()` / `setTemp_v(float temp_v)`: Gets or sets the temporary coin value.
3. `getPrice()` / `setPrice(float price)`: Gets or sets the final price of the drink.
4. `getCf()` / `setCf(float cf)`: Gets or sets the current credit inserted by the user.

DS2 :-

Purpose:
The `DS2` class is a concrete subclass of `DataStore` tailored for VM2. It is responsible for storing and managing the runtime data required by VM2 during its operations. Unlike `DS1` which uses `float`, `DS2` uses `int` values for all fields, aligning with the VM2 system design which handles integer-based transactions.

Member Variables:

1. `temp_p`: Temporarily holds the price of the drink before confirmation.
2. `temp_v`: Temporarily stores the value of the inserted coin.
3. `price`: Holds the confirmed, final price of the drink.

4. `cf`: Stores the current funds (credit) entered by the user.

Member Functions:

1. `getTemp_p()` / `setTemp_p(int temp_p)`: Retrieves or sets the temporary price.
2. `getTemp_v()` / `setTemp_v(int temp_v)`: Retrieves or sets the temporary value of a coin.
3. `getPrice()` / `setPrice(int price)`: Retrieves or sets the final drink price.
4. `getCf()` / `setCf(int cf)`: Retrieves or sets the current funds inserted.

State :-

Purpose:
The `State` class is an abstract base class that defines the interface for all concrete state classes in the vending machine system. It implements the State design pattern, where each concrete subclass overrides relevant methods to define behavior for a specific state (e.g., `Start`, `Idle`, `NoCups`, `CoinInserted`). The `StateContext` (in this system, likely `MDAEFSM`) interacts with this abstraction to manage transitions and delegate behavior.

Member Variables:

1. `A`: A static integer array, potentially used as shared temporary storage or configuration for state logic.
2. `op`: A reference to the `OP` class, which encapsulates all operations and business logic.

Member Functions:

1. `create()`: To be overridden by child states to handle the creation event.
2. `coin(int x)`: To handle coin insertion in the state-specific context.
3. `card()`: To process card payments in applicable states.
4. `setPrice()`: To handle setting the drink price.
5. `cancel()`: To handle canceling the current transaction.
6. `additive(int a)`: To handle adding sugar, cream, etc., as an additive.
7. `disposeDrink(int d)`: To trigger drink dispensing based on selection.
8. `printLine()`: Utility method to print a separator line, typically used for debugging or logging purposes.

Start :-

Purpose:
The `Start` class is a concrete implementation of the `State` class, representing the initial state in the MDA-based vending machine system. It is responsible for handling the system setup, particularly storing the initial drink price.

Member Variables:

1. `op`: Inherited from `State`, this reference enables the `Start` state to execute operations.
2. `A`: Inherited static array from `State`, re-initialized here but not used directly in this class.

Member Functions:

1. `Start(OP op)`: Constructor that initializes the operations handler and the static array.
2. `create()`: Overrides the abstract method to store the price by calling `op.storePrice()`, and provides visual feedback via `printLine()`.

NoCups :-

Purpose:
The `NoCups` class represents the state when the vending machine has no cups available for dispensing drinks. It is a concrete subclass of `State` and overrides specific behavior for handling coin input when the machine is in this condition.

Member Variables:

1. `op`: Inherited from `State`, used to perform operations such as increasing credit and returning coins.

Member Functions:

1. `NoCups(OP op)`: Constructor that initializes the operations handler.

2. `coin(int x)`: Overrides the method to handle coin insertion by increasing the credit and returning all coins, since drinks cannot be dispensed without cups.

Idle :-

Purpose:
The `Idle` class represents the state when the vending machine is ready for customer interaction but no payment has been completed yet. It handles coin insertion, card input, and price setting actions.

Member Variables:

1. `op`: Inherited from `State`, used to invoke operations such as storing price, increasing funds, and refunding coins.

Member Functions:

1. `Idle(OP op)`: Constructor that initializes the `OP` operations handler.
2. `coin(int x)`: Increases the current funds when a coin is inserted.
3. `card()`: Processes card input, refunds any coins, resets additives and credit.
4. `setPrice()`: Stores a new drink price using the `OP` handler.

CoinInserted :-

Purpose:
The `CoinInserted` class models the state in which the user has inserted sufficient payment (via coins or card). It allows further actions such as selecting additives, dispensing the drink, or canceling the transaction.

Member Variables:

1. `op`: Inherited from `State`; used to invoke various system operations.
2. `A`: Inherited static array used to track the selection status of additives.

Member Functions:

1. `CoinInserted(OP op)`: Constructor that initializes the `OP` handler.
2. `coin(int x)`: Increases current credit and returns change if overpaid.
3. `additive(int a)`: Toggles the selection of an additive.

4. `disposeDrink(int d)`: Dispenses additives and drink, returns any change, and resets funds and additive selections.
5. `cancel()`: Cancels the transaction, refunds all coins, and resets the credit and additives.

Strategy Pattern

AbstractFactory :-

Purpose:
The `AbstractFactory` class defines an interface for creating related components used by the vending machine system, including strategies for operations like storing prices, handling coins, and managing drinks or additives. It enables concrete factories (such as `ConcreteFactory1`, `ConcreteFactory2`) to provide specific implementations suitable for different machine types.

Member Functions:

1. `getDataStore()`: Returns the appropriate `DataStore` object (e.g., `DS1` or `DS2`).
2. `getDisposeAdditive()`: Provides the strategy to handle disposal of additives.
3. `getDisposeDrink()`: Provides the strategy to handle disposal of drinks.
4. `getReturnCoins()`: Provides the strategy for returning inserted coins (either fully or as change).
5. `getZeroCF()`: Provides the strategy for resetting the current funds.
6. `getStorePrice()`: Provides the strategy for storing the price of the selected drink.
7. `getIncreaseCF()`: Provides the strategy to increase the current funds (credit) based on input.

ConcreteFactory1 :-

Purpose:
The `ConcreteFactory1` class is the concrete implementation of the `AbstractFactory` for Vending Machine 1 (VM1). It provides specific strategy and data store objects (`DS1` and related strategies) that encapsulate the business logic and data handling required for VM1's operation.

Member Variables:

1. `dataStore`: Instance of `DS1`, the concrete data store implementation used by VM1.

Member Functions:

1. `ConcreteFactory1()`: Constructor that initializes the `DS1` data store for VM1.
2. `getDataStore()`: Returns the `DS1` instance.
3. `getDisposeAdditive()`: Returns `DisposeAdditive1`, the strategy for handling additives in VM1.
4. `getDisposeDrink()`: Returns `DisposeDrink1`, the strategy for dispensing drinks.
5. `getReturnCoins()`: Returns `ReturnCoins1`, the coin return strategy for VM1.
6. `getZeroCF()`: Returns `ZeroCF1`, the strategy for zeroing current funds.
7. `getStorePrice()`: Returns `StorePrice1`, the strategy for storing the drink price.
8. `getIncreaseCF()`: Returns `IncreaseCF1`, the strategy for increasing credit.

ConcreteFactory2 :-

Purpose:
The `ConcreteFactory2` class is a concrete implementation of the `AbstractFactory` used for VM2. It provides VM2-specific instances of all required strategies and data handling components. These objects encapsulate the core behaviors like storing prices, handling coins, and managing drink/additive disposal.

Member Variables:

1. `dataStore`: An instance of `DS2`, the data store implementation tailored for VM2.

Member Functions:

1. `ConcreteFactory2()`: Constructor that initializes the factory with a `DS2` instance.
2. `getDataStore()`: Returns the `DS2` instance.
3. `getDisposeAdditive()`: Returns the `DisposeAdditive2` strategy for VM2.
4. `getDisposeDrink()`: Returns the `DisposeDrink2` strategy for VM2.
5. `getReturnCoins()`: Returns the `ReturnCoins2` strategy for handling refunds in VM2.

6. `getZeroCF()`: Returns the `ZeroCF2` strategy to reset credit.
7. `getStorePrice()`: Returns the `StorePrice2` strategy to store the selected drink price.
8. `getIncreaseCF()`: Returns the `IncreaseCF2` strategy to update the credit based on inserted coins.

Strategy

DisposeAdditive :-

Purpose:
The `DisposeAdditive` class defines an abstract strategy for disposing of selected drink additives (such as sugar or cream). It is part of the Strategy design pattern and is intended to be subclassed by concrete implementations (`DisposeAdditive1`, `DisposeAdditive2`) for specific vending machine variants.

Member Variables:

1. `dataStore`: Protected reference to a `DataStore` object that allows access to relevant data needed during execution.

Member Functions:

1. `disposeAdditive(int[] A)`: Abstract method to be implemented by subclasses to define how additives are handled. The input array `A` represents additive selections (e.g., index 0 for sugar, index 1 for cream).

DisposeDrink :-

Purpose:
The `DisposeDrink` class is an abstract strategy used in the vending machine system to define the interface for dispensing drinks. Each concrete implementation (e.g., `DisposeDrink1`, `DisposeDrink2`) will implement the `disposeDrink()` method to handle vending logic appropriate to a specific machine.

Member Variables:

1. `dataStore`: A protected reference to the `DataStore` used to retrieve and manage machine-specific data during the drink dispensing process.

Member Functions:

1. `disposeDrink(int d)`: Abstract method to be overridden in concrete classes. The parameter `d` is typically an index or code representing the selected drink (e.g., 0 for coffee, 1 for chocolate).

IncreaseCF :-

Purpose:
The `IncreaseCF` class defines an abstract strategy for increasing the current funds (credit) in the vending machine system. Concrete subclasses implement the actual logic for how credit should be updated, depending on the data format and requirements of the specific machine (e.g., float for VM1, int for VM2).

Member Variables:

1. `dataStore`: A protected reference to the `DataStore`, used to access and modify the current funds.

Member Functions:

1. `increaseCF()`: Abstract method to be implemented by concrete strategy classes. It updates the current credit/funds, typically by adding the value of the most recently inserted coin.

ReturnCoins :-

Purpose:
The `ReturnCoins` class defines an abstract strategy for handling the return of coins in the vending machine system. This includes both returning change after a successful transaction and returning all inserted coins if the user cancels. The actual behavior is implemented in concrete subclasses (e.g., `ReturnCoins1`, `ReturnCoins2`).

Member Variables:

1. `dataStore`: A protected reference to the machine's `DataStore`, used to retrieve and reset credit information.

Member Functions:

1. `returnCoinsChange()`: Abstract method to return only the change left over after purchasing a drink.
2. `returnCoinsAll()`: Abstract method to return all inserted coins, usually when a transaction is cancelled.

StorePrice :-

Purpose:
The `StorePrice` class is an abstract strategy that defines the interface for storing the price of a drink in the vending machine system. Concrete implementations (such as `StorePrice1` or `StorePrice2`) provide the logic specific to the data format and requirements of each vending machine variant.

Member Variables:

1. `dataStore`: A protected reference to the `DataStore`, used to store or retrieve the drink price.

Member Functions:

1. `storePrice()`: Abstract method to be implemented by subclasses to save the temporary price (`temp_p`) as the final price in the data store.

ZeroCF :-

Purpose:
The `ZeroCF` class defines an abstract strategy for resetting the current funds (credit) in the vending machine system to zero. It is typically used after a transaction completes or when the user cancels. Concrete subclasses provide implementation specific to the machine's data type and storage format.

Member Variables:

1. `dataStore`: A protected reference to the `DataStore`, used to access and reset the credit value.

Member Functions:

1. `zeroCF()`: Abstract method to be implemented in concrete classes to reset the `cf` (current funds) field in the data store.

DisposeAdditive1 :-

Purpose:
The `DisposeAdditive1` class is a concrete implementation of the `DisposeAdditive` strategy for VM1. It handles the disposal of drink additives—specifically sugar, which is the only supported additive in this version.

Member Variables:

1. `dataStore`: Inherited from `DisposeAdditive`, used to interface with VM1-specific data if needed (not directly used in this method).

Member Functions:

1. `DisposeAdditive1(DataStore dataStore)`: Constructor that assigns the associated `DataStore`.
2. `disposeAdditive(int[] A)`: Prints confirmation of selected additive(s). Currently supports only sugar, which corresponds to index `0` in the array `A`.

DisposeDrink1 :-

Purpose:
The `DisposeDrink1` class is a concrete implementation of the `DisposeDrink` strategy for VM1. It defines how specific drinks—cappuccino and chocolate—are dispensed based on user selection.

Member Variables:

1. `dataStore`: Inherited from `DisposeDrink`, used to interface with VM1's data (not directly used in this method).

Member Functions:

1. `DisposeDrink1(DataStore dataStore)`: Constructor that sets the associated data store.
2. `disposeDrink(int d)`: Prints the corresponding drink being disposed.
   Accepts:
   - `0` for cappuccino
   - `1` for chocolate
   - Any other value results in an error message.

IncreaseCF1 :-

Purpose:
The `IncreaseCF1` class provides the concrete implementation of the `IncreaseCF` strategy for VM1. It updates the current credit (`cf`) by adding the most recently inserted coin value (`temp_v`), which is represented as a float in VM1.

Member Variables:

1. `dataStore`: Inherited from `IncreaseCF`; cast to `DS1` to allow access to float-specific getters and setters.

Member Functions:

1. `IncreaseCF1(DataStore dataStore)`: Constructor that assigns the associated `DS1` instance.
2. `increaseCF()`: Adds `temp_v` to `cf` and prints the new total credit.

ReturnCoins1 :-

Purpose:
The `ReturnCoins1` class provides the coin return strategy implementation for VM1. It handles both returning all inserted coins (typically during cancellation) and returning only the excess coins when the inserted amount exceeds the drink price.

Member Variables:

1. `dataStore`: Inherited from `ReturnCoins`; cast to `DS1` for float-specific operations.

Member Functions:

1. `ReturnCoins1(DataStore dataStore)`: Constructor that sets the `DS1` data store instance.
2. `returnCoinsAll()`: Checks if there is any credit (`cf`) and returns the entire amount, then resets `cf` to 0.
3. `returnCoinsChange()`: Calculates and returns the change (credit minus price), then sets `cf` to match the drink price.

StorePrice1 :-

Purpose:
The `StorePrice1` class provides the concrete implementation of the `StorePrice` strategy for VM1. It stores the drink price by transferring the temporary price (`temp_p`) into the permanent field (`price`) and confirms the update via output.

Member Variables:

1. `dataStore`: Inherited from `StorePrice`; cast to `DS1` to access float-based pricing fields.

Member Functions:

1. `StorePrice1(DataStore dataStore)`: Constructor that initializes the class with VM1's data store.
2. `storePrice()`: Copies the temporary price (`temp_p`) into the permanent `price` field and prints a confirmation message.

ZeroCF1 :-

Purpose:
The `ZeroCF1` class provides the concrete implementation of the `ZeroCF` strategy for VM1. It is responsible for resetting the user's current credit (`cf`) to zero, typically after a transaction is completed or cancelled.

Member Variables:

1. `dataStore`: Inherited from `ZeroCF`; cast to `DS1` to work with VM1's float-based credit system.

Member Functions:

1. `ZeroCF1(DataStore dataStore)`: Constructor that sets the data store for VM1.
2. `zeroCF()`: Resets the `cf` field to 0 and prints a confirmation message.

DisposeAdditive2 :-

Purpose:
The `DisposeAdditive2` class is a concrete implementation of the `DisposeAdditive` strategy for VM2. It handles the selection and confirmation of additives—specifically sugar and cream—when preparing a drink.

Member Variables:

1. `dataStore`: Inherited from `DisposeAdditive`; assigned via constructor. Not used directly in this method but retained for consistency across strategy classes.

Member Functions:

1. `DisposeAdditive2(DataStore dataStore)`: Constructor that sets the data store.
2. `disposeAdditive(int[] A)`: Prints a confirmation message listing all selected additives.
    - `A[0] == 1`: Sugar selected
    - `A[1] == 1`: Cream selected

DisposeDrink2 :-

Purpose:
The `DisposeDrink2` class provides the concrete implementation of

the `DisposeDrink` strategy for VM2. It supports the dispensing of only one drink option: coffee.

Member Variables:

1. `dataStore`: Inherited from `DisposeDrink`; assigned in the constructor. Not directly used in this method.

Member Functions:

1. `DisposeDrink2(DataStore dataStore)`: Constructor that assigns the `DS2` data store.
2. `disposeDrink(int d)`: Dispenses coffee if `d == 0`; otherwise, prints an invalid option message.

IncreaseCF2 :-

Purpose:
The `IncreaseCF2` class implements the `IncreaseCF` strategy for VM2. It updates the user's credit (`cf`) by adding the value of the most recently inserted coin (`temp_v`). This version uses integer arithmetic appropriate for VM2.

Member Variables:

1. `dataStore`: Inherited from `IncreaseCF`; cast to `DS2` to use integer-based fields.

Member Functions:

1. `IncreaseCF2(DataStore dataStore)`: Constructor that sets the data store for VM2.
2. `increaseCF()`: Adds the temporary coin value to the current funds and prints the updated credit.

ReturnCoins2 :-

Purpose:
The `ReturnCoins2` class is a concrete implementation of the `ReturnCoins` strategy for

VM2. It provides logic to return all inserted coins upon cancellation or to return just the excess coins if the credit exceeds the drink price.

Member Variables:

1. `dataStore`: Inherited from `ReturnCoins`; cast to `DS2` to support integer-based values used by VM2.

Member Functions:

1. `ReturnCoins2(DataStore dataStore)`: Constructor that assigns the VM2-specific data store.
2. `returnCoinsAll()`: Returns the entire credit and resets the credit (`cf`) to zero.
3. `returnCoinsChange()`: Returns only the excess amount and updates `cf` to match the price.

StorePrice2 :-

Purpose:
The `StorePrice2` class provides the concrete implementation of the `StorePrice` strategy for VM2. It updates the final drink price in the data store by copying it from the temporary price field.

Member Variables:

1. `dataStore`: Inherited from `StorePrice`; cast to `DS2` to work with VM2's integer pricing system.

Member Functions:

1. `StorePrice2(DataStore dataStore)`: Constructor that sets the data store for VM2.
2. `storePrice()`: Copies the temporary price (`temp_p`) to the permanent price field and prints a confirmation message.

ZeroCF2 :-

Purpose:
The `ZeroCF2` class is a concrete implementation of the `ZeroCF` strategy for VM2. It is responsible for resetting the current funds (`cf`) to zero, typically after a drink is dispensed or a transaction is canceled.

Member Variables:

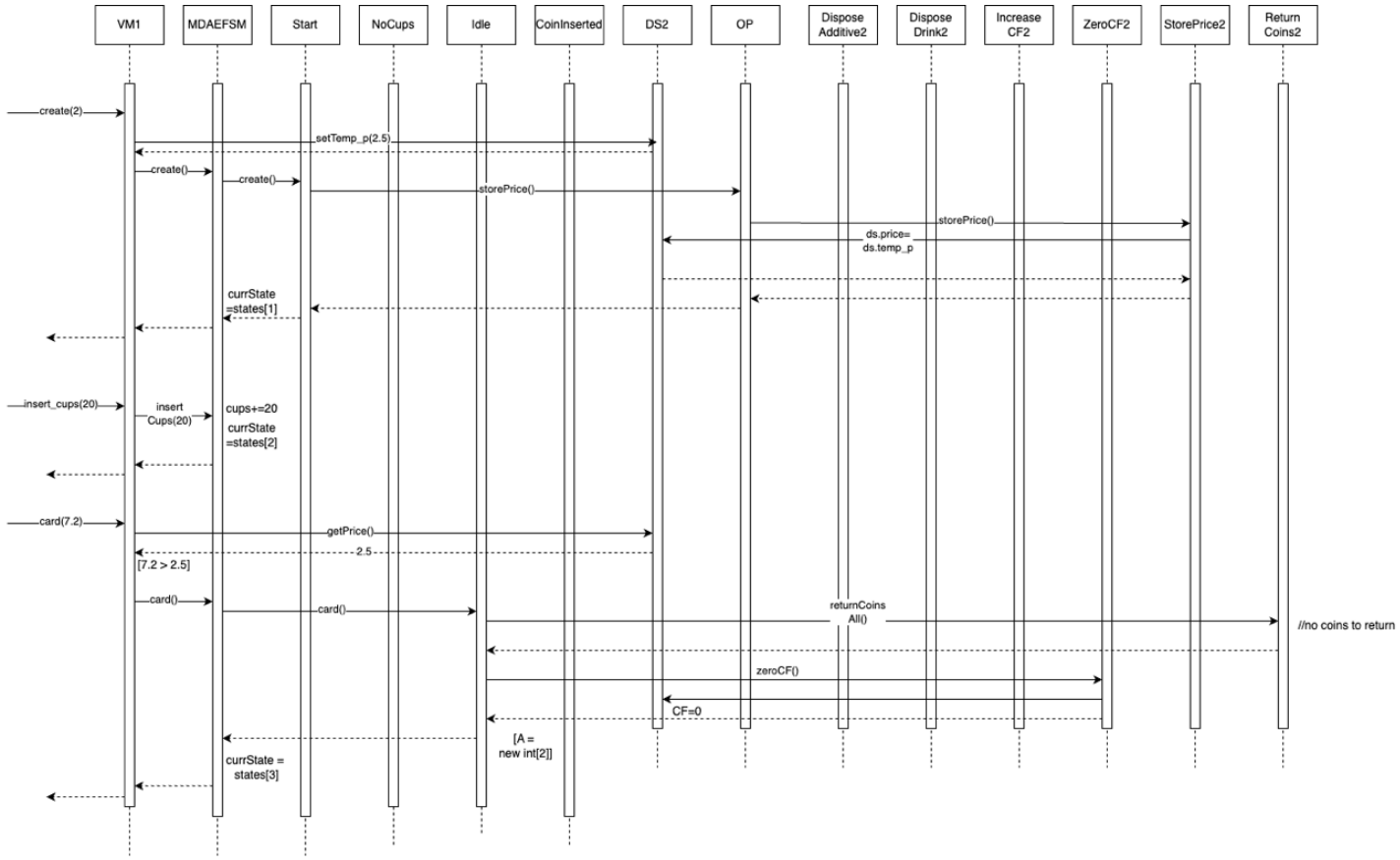
1. `dataStore`: Inherited from `ZeroCF`; cast to `DS2` to handle integer-based operations in VM2.

Member Functions:

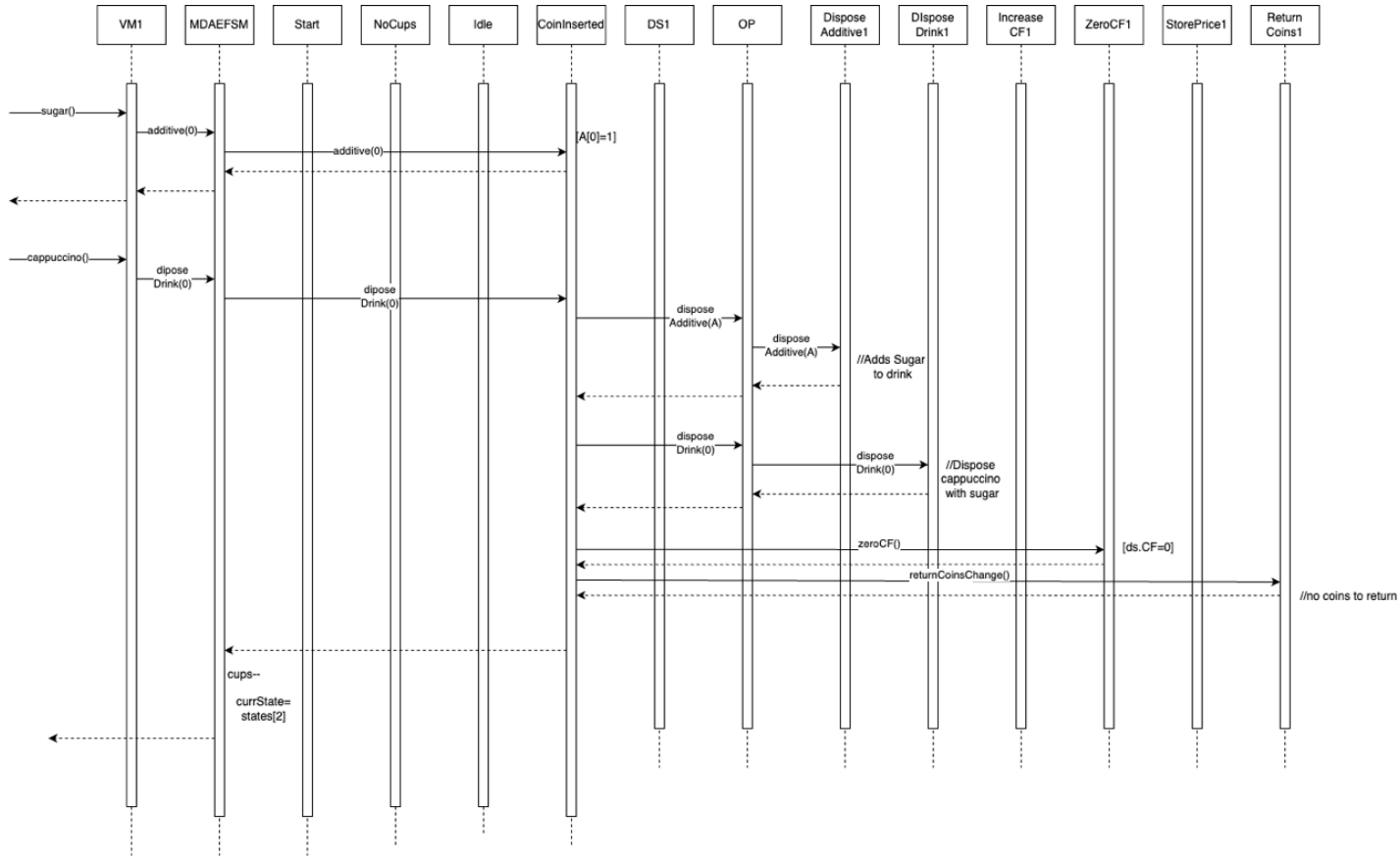
1. `ZeroCF2(DataStore dataStore)`: Constructor that assigns the VM2-specific data store.
2. `zeroCF()`: Resets the credit (`cf`) to zero and prints a confirmation message.

Sequence Diagram 1

1) create(2.5), insert_cups(20), card(7.2), sugar(), cappuccino()



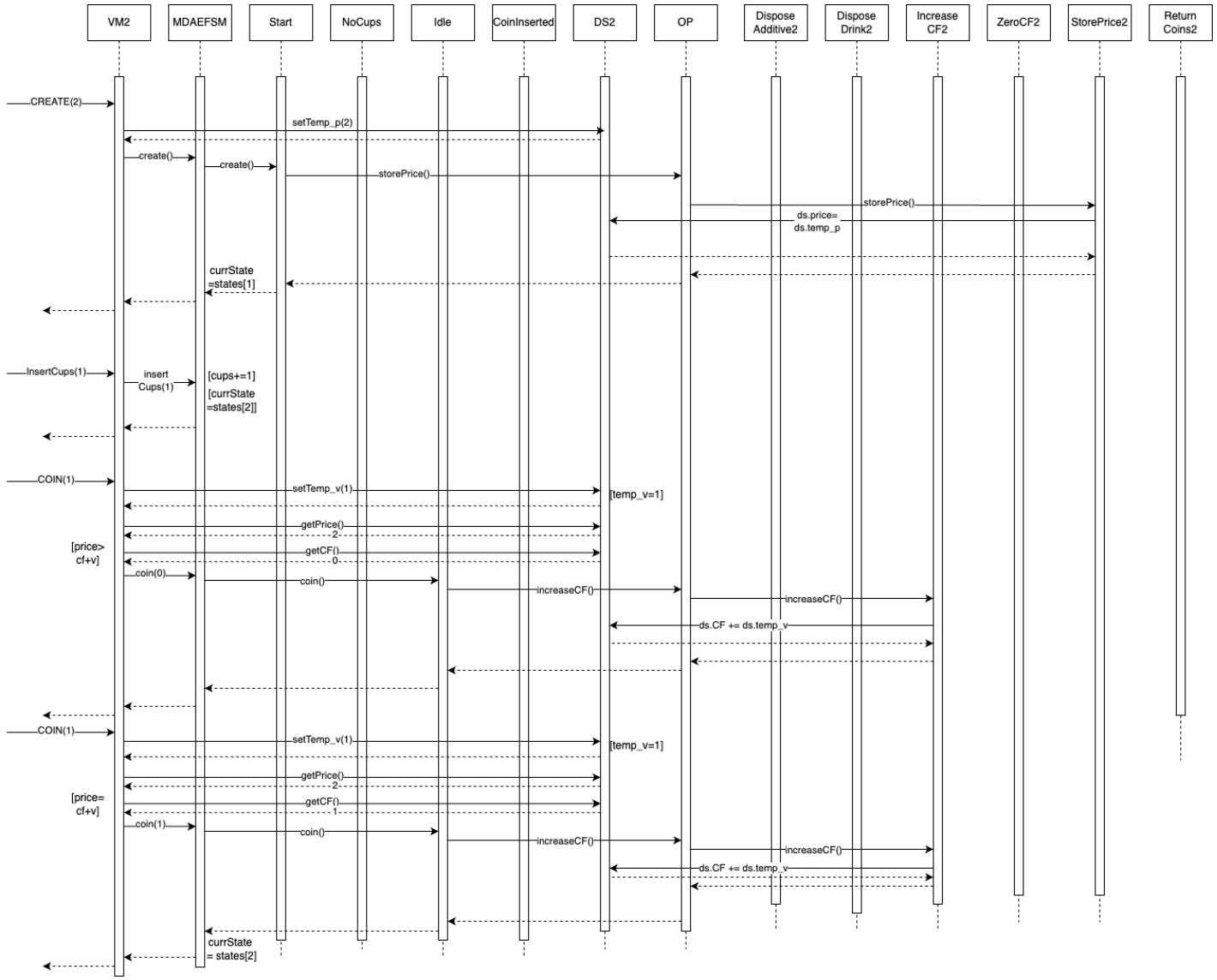
Part2) continued


VM1
MDAEFSM
Start
NoCups
Idle
CoinInserted
DS1
OP
Dispose Additive1
DIspose Drink1
Increase CF1
ZeroCF1
StorePrice1
Return Coins1
sugar()
additive(0)
additive(0)
[A[0]=1]
cappuccino()
dipose Drink(0)
dipose Drink(0)
dispose Additive(A)
dispose Additive(A)
//Adds Sugar to drink
dispose Drink(0)
dispose Drink(0)
//Dispose cappuccino with sugar
zeroCF()
[ds.CF=0]
returnCoinsChange()
//no coins to return
cups--
currState= states[2]

Sequence Diagram 2)

:

CREATE(2), InsertCups(1), COIN(1), COIN(1), CREAM(), COFFEE()



Part2)Continued