Class Diagram for the Vending Machine

Learn to create a class diagram for the vending machine using the bottom-up approach.

We'll cover the following

- Components of a vending machine
  - State
  - Product
  - Rack
  - Inventory
  - Vending machine
  - Enumeration
    - ProductType
- Relationship between the classes
  - Composition
  - Aggregation
  - Association
  - Inheritance
- Class diagram for the vending machine
- Design pattern
- AI-powered trainer
- Additional requirements

In this lesson, we'll identify and design the classes and interfaces based on the requirements that we have previously gathered from the interviewer for our vending machine system.

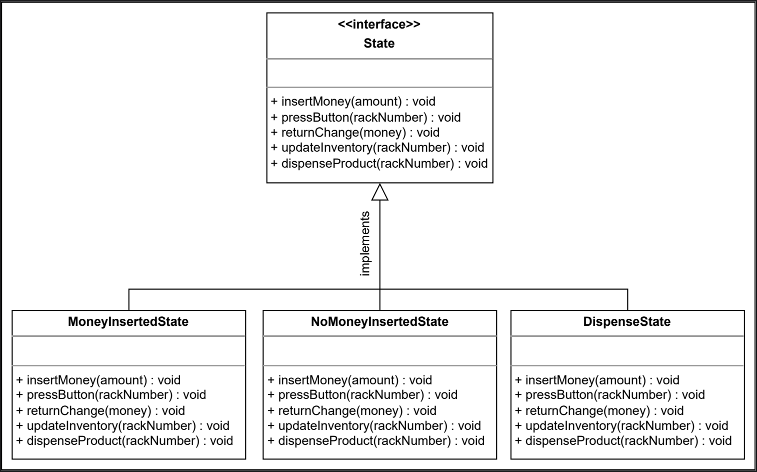
Components of a vending machine

As mentioned earlier, we should design the vending machine using a bottom-up approach.

State

`State` is an interface that represents the current state of the vending machine. There can be three possible states of a vending machine, i.e., no money inserted state, money inserted state, and the dispense state. The `NoMoneyInsertedState` class represents the state when there is no money inserted in the machine. When the user inserts the money into the machine, the state changes to `MoneyInsertedState`. Furthermore, when the machine dispenses the required product to the user, it transitions to the `DispenseState`.

This problem follows the State design pattern since the vending machine changes its behavior based on its state. Here, the `State` class defines an interface for declaring what the subclasses (`NoMoneyInsertedState`, `MoneyInsertedState`, `DispenseState`) should do. The subclasses provide the implementation for methods defined in the `State`, and the implementation of each method changes with the change of the state. Every state implements some functions, as shown below:


<<interface>>
State
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
implements
MoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
NoMoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
DispenseState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void

The class diagram of the State interface and its subclasses

R2: Vending Machine

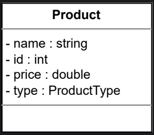
R2: The vending machine can be in one of these three states:

- There is no money inserted into the machine.
- Money is inserted into the machine.
- The machine gives out the product.

> **Note:** According to the implementation of the State design pattern, all functions will be available in each state. However, it is not necessary that every function has a meaningful definition for that particular state.

Product

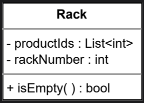
The `Product` class contains the details of a particular product available in the vending machine. Each product has `name`, `id`, `price`, and its `type` associated with it as presented below:


Product
- name : string
- id : int
- price : double
- type : ProductType

The class diagram of the Product class

Rack

The `Rack` class is used to identify the location of the products in the vending machine. Every rack has a specific `rackNumber` as an identifier. The class representation is provided below:


Rack
- productIds : List<int>
- rackNumber : int
+ isEmpty( ) : bool

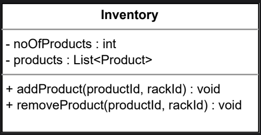
The class diagram of the Rack class

R1: Vending Machine

R1: There are different products placed at different positions in the vending machine.

Inventory

The `Inventory` class will contain a list of products available at different positions inside the vending machine. It will also reference the name of the product present in the particular rack. This class is also responsible for adding a product to the vending machine. The class representation is provided below:


Inventory
- noOfProducts : int
- products : List<Product>
+ addProduct(productId, rackId) : void
+ removeProduct(productId, rackId) : void

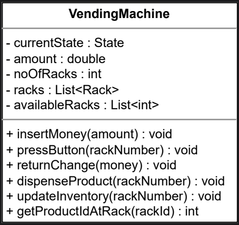
The class diagram of the Inventory class

R4: Vending Machine

R4: The admin can add a product to the machine or remove a product from the machine.

Vending machine

`VendingMachine` is a class that represents the whole vending machine. The `State` type variable is used to define the current state of the vending machine. The vending machine has a list of `racks` and `amount` stored in it. This class follows the Singleton design pattern, since there will only be one instance of the class. The class is presented below:


VendingMachine
- currentState : State
- amount : double
- noOfRacks : int
- racks : List<Rack>
- availableRacks : List<int>
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ dispenseProduct(rackNumber) : void
+ updateInventory(rackNumber) : void
+ getProductIdAtRack(rackId) : int

The class diagram of the VendingMachine class

R6, R8, R9, and R10: Vending Machine

R6: The user can insert money into the machine in the form of cash.

R8: The system should check whether the user has inserted the exact amount required for the specific product into the machine.

R9: If the amount is greater than the product price, the system should change back the user and dispense the product.

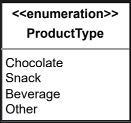
R10: If the amount is less than the product price, the system should display an error message, and return the money.

Enumeration

The enumeration required to design the vending machine system is provided below:

ProductType

We need to create an enumeration to keep track of the type of product, whether it is a chocolate, snack, beverage, or other.


<<enumeration>>
ProductType
Chocolate
Snack
Beverage
Other

The ProductType enumeration

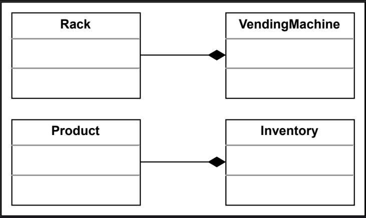
Relationship between the classes

Now, we’ll discuss the relationships between the classes we have defined above in our vending machine.

Composition

The class diagram has the following composition relationships.

- The `VendingMachine` class is composed of the `Rack` class.
- The `Inventory` class is composed of the `Product` class.

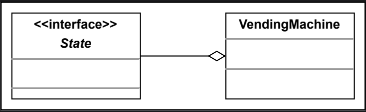

Rack
VendingMachine
Product
Inventory

The composition relationship between classes

Aggregation

The class diagram has the following aggregation relationships:

- The `VendingMachine` class contains the `State` interface.


<<interface>>
State
VendingMachine

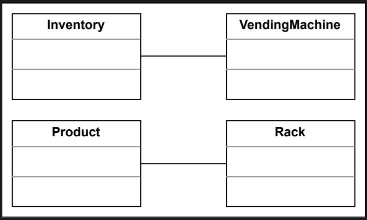
The aggregation relationship between classes

Association

The class diagram has the following association relationships.

- The `VendingMachine` class has a two-way association with the `Inventory` class.

- The `Rack` class has a two-way association with the `Product` class.


Inventory
VendingMachine
Product
Rack

The association relationship between classes

Inheritance

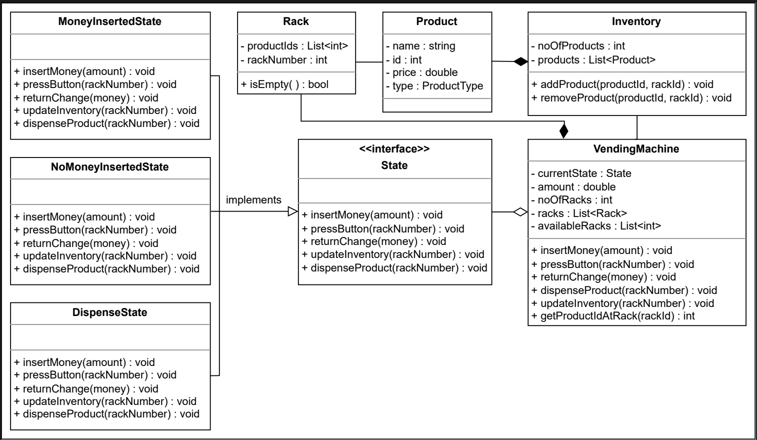
The following classes show an inheritance relationship:

- The `NoMoneyInsertedState`, `MoneyInsertedState`, and `DispenseState` classes implement the `State` interface.

> **Note:** We have already discussed this inheritance relationship between classes in the component section above.

Class diagram for the vending machine

Here is the complete class diagram for our vending machine:


MoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
Rack
- productIds : List<int>
- rackNumber : int
+ isEmpty( ) : bool
Product
- name : string
- id : int
- price : double
- type : ProductType
Inventory
- noOfProducts : int
- products : List<Product>
+ addProduct(productId, rackId) : void
+ removeProduct(productId, rackId) : void
NoMoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
implements
<<interface>>
State
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
VendingMachine
- currentState : State
- amount : double
- noOfRacks : int
- racks : List<Rack>
- availableRacks : List<int>
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ dispenseProduct(rackNumber) : void
+ updateInventory(rackNumber) : void
+ getProductIdAtRack(rackId) : int
DispenseState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void

The class diagram of the vending machine

Design pattern

We have used the State design pattern to design this problem because, in different states, we perform different or specific tasks according to the state. The vending machine changes its behavior based on its state. The different states within the system are listed below:

- No money inserted state
- Money inserted state
- Dispense state

All these states have the same methods but the implementation of each method in each state changes with the change of the state.

AI-powered trainer

At this stage, everything should be clear. If you encounter any confusion or ambiguity, feel free to utilize the interactive AI-enabled widget below to seek clarification. This tool is designed to assist you in strengthening your understanding of the concepts.

Powered by AI

20 Prompts Remaining

Prompt AI Widget

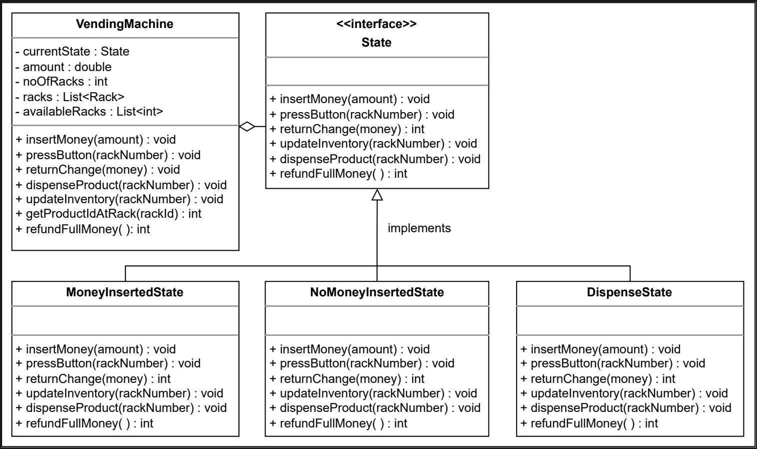
Our tool is designed to help you to understand concepts and ask any follow up questions. Ask a question to get started.

Enter Prompt Here...

Additional requirements

The interviewer can introduce some additional requirements in the vending machine, or they can ask some follow-up questions. Let's see some examples of additional requirements:

Refund/Cancel: The vending machine should have the option to cancel the operation. In that case, the customers will get a refund. The class diagram provided below shows the refund functionality in all states:


VendingMachine
- currentState : State
- amount : double
- noOfRacks : int
- racks : List<Rack>
- availableRacks : List<int>
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : void
+ dispenseProduct(rackNumber) : void
+ updateInventory(rackNumber) : void
+ getProductIdAtRack(rackId) : int
+ refundFullMoney( ): int
<<interface>>
State
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : int
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
+ refundFullMoney( ) : int
implements
MoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : int
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
+ refundFullMoney( ) : int
NoMoneyInsertedState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : int
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
+ refundFullMoney( ) : int
DispenseState
+ insertMoney(amount) : void
+ pressButton(rackNumber) : void
+ returnChange(money) : int
+ updateInventory(rackNumber) : void
+ dispenseProduct(rackNumber) : void
+ refundFullMoney( ) : int

Modified class diagram of the vending machine

In the class diagram above, we can see a function, `refundFullMoney()`, in all states which is used to refund the full money. Here is the definition of the `refundFullMoney()` function according to each state:

- `NoMoneyInsertedState`: In this state, the `refundFullMoney()` function throws an exception or warning, “please insert some cash first” because we we have not inserted any money yet.
- `MoneyInsertedState`: In this state, we know the customer inserted the cash because, due to this, we shifted the state of the vending machine from `NoMoneyInsertedState` to `MoneyInsertedState`. If the customer uses the refund