CSE201 Assignment 3

-Due: 2024/11/15 (FRI) –

General Instruction

This is an independent assignment, so each student should submit their own answers without collaborating other students. Copying, cheating or any other academic misconduct is strictly prohibited.

There are two parts of the assignments: (Part 1) Programming Assignment, (Part 2) Written Assignment. For the programming assignment, you must submit source code, and (if applicable) written answers; for the written assignment, you only need to submit written answers.

The total grading points are **25pt**, and the points assigned for each question may be different.

Submission Instruction

For each question, create source code or text (word, hwp) files according to the question-specific instruction.

Compress all the files with the filename "**hw3_YourName_YourStudentID.zip**", and submit this single file.

Part 1. Programming Assignment (20 pt)

You will modify the program of Assignment 2 (i) to decouple some functionality from *VendingMachine* class, and (ii) to use the inheritance concept.

In Assignment 2, recall that *VendingMachine* implemented the following state-dependent actions: [*Insert Coin*], [*Eject Coin*], [*Dispense Product*]; that is, *State* class did not have many roles in terms of implementing FSM semantics other than keeping the state name. Now, we want to move the implementations of the state-dependent actions (previously implemented in *VendingMachine* class) to *State* class.

State class is extended to have pure virtual functions only (note that a class whose member functions consist of pure virtual functions only is called an *interface class*). In particular, [*insertCoin*], [*ejectCoin*], [*dispense*] are defined as pure virtual functions in *State* class; that is, these pure virtual functions must be implemented by its child classes via inheritance. Three child classes are derived from *State* class via inheritance: *NoCoinState, HasCoinState, SoldOutState*. Each child class implements the pure virtual functions defined in *State* class in a way to provide the same FSM semantics in Assignment 2.

More specifically, the three member functions [*insertCoin*], [*ejectCoin*], [*dispense*] in *VendingMachine* class, will simply call the corresponding *virtual functions* in *State* class; those calls eventually trigger the corresponding functions in each child class (i.e., *NoCoinState, HasCoinState, SoldOutState*). We provide the complete code of [*insertCoin*], [*ejectCoin*], [*dispense*] in *VendingMachine* class.

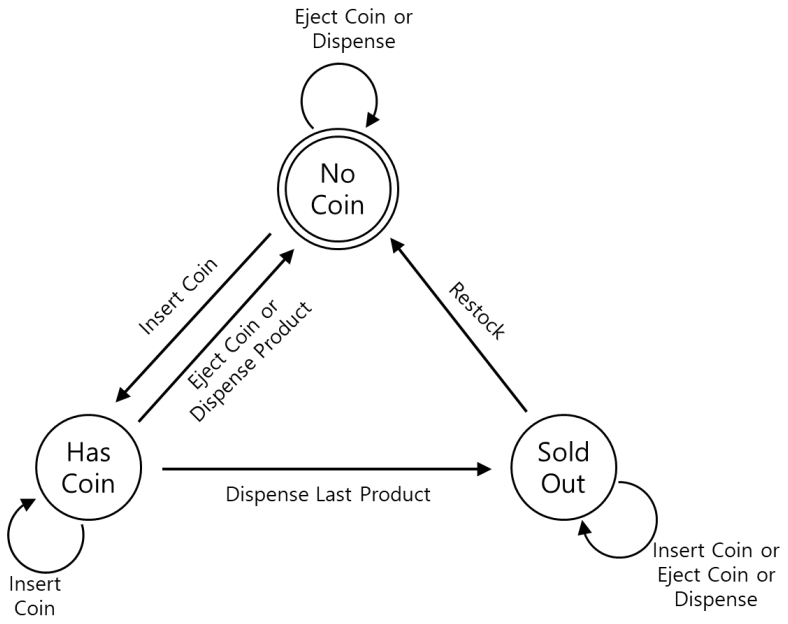
In addition, we also extend *Product* class to use the inheritance concept. Two child classes are defined to express the specialized feature: *Beverage* class and *Snack* class. These child classes override several member functions of its parent class (i.e., *Product* class).

You will extend the Assignment 2 according to the above explanation. We provide the class skeletons that you should implement to achieve this goal; you should modify some classes from Assignment 2 or add some new classes.

(**Note**) You may reuse many parts of source code from Assignment 2.

(**Note**) The output of Assignment 3 is almost identical to Assignment 2

(**Note**) We do not explain the member variables and functions that were already explained in the previous assignments.



(1) [**Product**] class

```
class Product {
protected:
    static int nextId;
    int id;
    string p_name;
    double p_price;
    double p_calorie;

public:
    Product(string name, double price, double p_calorie);
    virtual ~Product();
    virtual string getName() const { return p_name; }
    virtual double getPrice() const { return p_price; }
    virtual int getId() const { return id; }
    virtual void describe() const {
        cout << "Product: " << p_name << " (ID: " << id << ", Price: $" << p_price
<< ")\n";
    }
};
```

- *~Product()*:Virtual destructor to destroy the object.

- *getName()*: Virtual function to return the product's name

- *getPrice()*: Virtual function to return the product's price

- *getId()*: Virtual function to return the unique identifier for the product

- *describe()*: Virtual function to display a console message. Note that this function can be overridden by its child classes (e.g., Beverage, Snack classes) to display more specialized messages.

(2) [**Beverage**] class

```
class Beverage : public Product {
```

```
public:
    Beverage(string name, double price, double calorie);
    void describe() const override {
    cout << "Beverage: " << getName() << " (ID: " << getId()
           << ", Price: $" << getPrice() << ", Calories: " << this->p_calorie <<
"calories)\n";
    }
};
```

- *Beverage(string name, double price, double calorie)*:Constructor to initialize the object; name, price p_calories are copied to the corresponding member variables. It displays a console message starting "Beverage created…" See the details in the sample console output.

- *describe()*: Displays a console message as shown in the implementation.

(3) [**Snack**] class

```
class Snack : public Product {
public:
    Snack(string name, double price, double calorie);
    void describe() const override {
    cout << "Snack: " << getName() << " (ID: " << getId()
           << ", Price: $" << getPrice() << ", Calories: " << this->p_calorie <<
"calories)\n";
    }
};
```

- *Snack(string name, double price, double calorie):* Constructor to initialize the object; name, price p_calories are copied to the corresponding member variables. It displays a console message starting "Snack created…" See the details in the sample console output.

- *describe()*: Displays a console message as shown in the implementation.

(4) [**State**] class

```
class State {
protected:
    string s_name;
    VendingMachine* machine;

public:
    State(VendingMachine* machine);
    virtual string getName() const = 0;
    virtual ~State();
    virtual void insertCoin(double coin) = 0;
    virtual void ejectCoin() = 0;
    virtual void dispense(string productName) = 0;
};
```

-*VendingMachine* machine*: Pointer to the associated *VendingMachine*

-*State(VendingMachine* machine)*: Constructor to initialize the object. The pointer to *VendingMachine* is passed as an argument.

- *getName()*: Virtual function to return the name of the state

- *~State()*: Virtual destructor; it displays a console message starting "[Destructor]…" along with its state name. See the details in the sample console output.

- *insertCoin(double coin)*: Virtual function to handle the action [*Insert Coin*]. Its implementation at child classes (*i.e., NoCoinState, HasCoinState, SoldOutState*) differ depending on the state.

- *ejectCoin()*: Virtual function to handle the action [*Eject Coin*]. Its implementation at child classes (*i.e., NoCoinState, HasCoinState, SoldOutState*) differ depending on the state.

- *dispense(string productName)*: Virtual function to handle the action [*Dispense*]. Its implementation at child classes (*i.e., NoCoinState, HasCoinState, SoldOutState*) differ depending on the state.

(5) [**NoCoinState**] class

```
class NoCoinState : public State {
public:
    NoCoinState(VendingMachine* t_machine);
    ~NoCoinState() override;
    void insertCoin(double coin) override;
    void ejectCoin() override;
    void dispense(string productName) override;
    string getName() const override { return this->s_name; }
};
```

- *NoCoinState(VendingMachine* t_machine):* Constructor to initialize the object. The pointer to *VendingMachine* is passed as an argument. It displays a console message "[Constructor]…." along with the state name. See the details in the sample console output.

-*~NoCoinState()*: Destructor to destroy the object.

- *insertCoin(double coin)*: A function that overrides *insertCoin* in *State* class. It implements the state-specific action for [*No Coin*] state.

- *ejectCoin()*: A function that overrides *ejectCoin* in *State* class. It implements the state-specific action for [*No Coin*] state.

- *dispense(string productName)*: A function that overrides *dispense* in *State* class. It implements the state-specific action for [*No Coin*] state.

- *getName()*: A function that overrides *getName* in *State* class. It returns the name of the state.

(6) [**HasCoinState**] class

```
class HasCoinState : public State {
public:
    HasCoinState(VendingMachine* t_machine);
    ~HasCoinState() override;
    void insertCoin(double coin) override;
    void ejectCoin() override;
    void dispense(string productName) override;
    string getName() const override { return this->s_name; }
```

```
};
```

- *HasCoinState(VendingMachine* t_machine)*: Constructor to initialize the object. The pointer to *VendingMachine* is passed as an argument. It displays a console message "[Constructor]…." along with the state name. See the details in the sample console output.

-*~HasCoinState()*: Destructor to destroy the object.

- *insertCoin(double coin)*: A function that overrides *insertCoin* in *State* class. It implements the state-specific action for [*Has Coin*] state.

- *ejectCoin()*: A function that overrides *ejectCoin* in *State* class. It implements the state-specific action for [*Has Coin*] state.

- *dispense(string productName)*: A function that overrides *dispense* in *State* class. It implements the state-specific action for [*Has Coin*] state.

- *getName()*: A function that overrides *getName* in *State* class. It returns the name of the state.

(7) [**SoldOutState**] class

```
class SoldOutState : public State {
public:
    SoldOutState(VendingMachine* t_machine);
    ~SoldOutState() override;
    void insertCoin(double coins) override;
    void ejectCoin() override;
    void dispense(string productName) override;
    string getName() const override { return this->s_name; }
};
```

- *SoldOutState(VendingMachine* t_machine)*: Constructor to initialize the object. The pointer to *VendingMachine* is passed as an argument. It displays a console message "[Constructor]…." along with the state name. See the details in the sample console output.

-*~SoldOutState()*: Destructor to destroy the object.

- *insertCoin(double coins)*: A function that overrides *insertCoin* in *State* class. It implements the state-specific action for [*Sold Out*] state.

- *ejectCoin()*: A function that overrides *ejectCoin* in *State* class. It implements the state-specific action for [*Sold Out*] state.

- *dispense(string productName)*: A function that overrides *dispense* in *State* class. It implements the state-specific action for [*Sold Out*] state.

- *getName()*: A function that overrides *getName* in *State* class. It returns the name of the state.

(8) [**VendingMachine**] class

```
class VendingMachine {
private:
    State* noCoinState;
    State* hasCoinState;
```

```
    State* soldOutState;

    State* currentState;
    const int MAX_NUM_PRODUCT = 10;
    Product* inventory[10];
    int num_of_products;
    bool hasCoin;
    double coinValue;

    void printState(string action) const {
        cout << "Action: " << action << " | Current State: " << currentState-
>getName() << " | Coin Value: " << this->coinValue << "\n";
    }

public:
    VendingMachine();
    ~VendingMachine();
    void setState(State* state);
    void insertCoin(double coin);
    void ejectCoin();
    void dispense(string productName);

    Product* removeProduct(string productName);
    void addProduct(Product* product);

    bool isProductAvailable(string productName) const;
    double getProductPrice(string productName) const;
    void displayInventory() const;
    int getInventoryCount() const { return num_of_products; }

    State* getNoCoinState() const { return noCoinState; }
    State* getHasCoinState() const { return hasCoinState; }
    State* getSoldOutState() const { return soldOutState; }

    bool hasInsertedCoin() const { return hasCoin; }
    void setCoinInserted(bool inserted) { hasCoin = inserted; }
    double getCoinValue() const { return coinValue; }
    void resetCoinValue() { coinValue = 0.0; }
    void addCoinValue(double coin) { coinValue += coin; }
};

void VendingMachine::insertCoin(double coin) {
    currentState->insertCoin(coin);
    printState("Insert Coin");
}

void VendingMachine::ejectCoin() {
    currentState->ejectCoin();
    printState("Eject Coin");
}

void VendingMachine::dispense(string productName) {
    currentState->dispense(productName);
    printState("Dispense");
}
```

- (**Note**) The explanations of the most member variables and functions are similar to Assignment 2.

- *void insertCoin(double coin):* A handler for the action [*Insert Coin*]. The state-specific behavior of the action is implemented in *State* and its derived classes (*i.e., NoCoinState, HasCoinState, SoldOutState*). That is, *VendingMachine* simply calls the corresponding action handler defined in *State* class, and we give a complete implementation in *VendingMachine*.

- *void ejectCoin():* A handler for the action [*Eject Coin*]. The state-specific behavior of the action is implemented in *State* and its derived classes (*i.e., NoCoinState, HasCoinState, SoldOutState*). That is, *VendingMachine* simply calls the corresponding action handler defined in *State* class, and we give a complete implementation in *VendingMachine*.

- *void dispense(string productName):* A handler for the action [*Dispense*]. The state-specific behavior of the action is implemented in *State* and its derived classes (*i.e., NoCoinState, HasCoinState, SoldOutState*). That is, *VendingMachine* simply calls the corresponding action handler defined in *State* class, and we give a complete implementation in *VendingMachine*.

(9) Example of Main function with a sample test routine

main function:

```
int main() {
    cout << "==========Part 1==========" << endl;
    VendingMachine machine; //Create a VendingMachine instance

    Product* p_cola = new Beverage("Cola", 1.50, 330);  //Create a new product
    Product* p_chips = new Snack("Chips", 1.00, 150); //Create a new product
    Product* p_water = new Beverage("Water", 1.00, 0); //Create a new product

    cout << "==========Part 2==========" << endl;
    //Add three products
    machine.addProduct(p_cola);
    machine.addProduct(p_chips);
    machine.addProduct(p_water);
    //Display inventory
    machine.displayInventory();

    cout << "==========Part 3==========" << endl;
    //Insert coin and dispense
    machine.insertCoin(1.00);
    machine.insertCoin(0.50);
    machine.dispense("Cola");

    cout << "==========Part 4==========" << endl;
    machine.insertCoin(1.00);
    machine.dispense("Water");

    cout << "==========Part 5==========" << endl;
    machine.insertCoin(0.50);
    machine.dispense("Chips"); // Should say insufficient funds

    machine.insertCoin(0.50);
    machine.dispense("Chips");

    cout << "==========Part 6==========" << endl;
    machine.insertCoin(1.00);
    machine.dispense("Candy"); // Should say product not available
```

```
    machine.displayInventory();

    cout << "==========Part 7==========" << endl;
    Product* p_chocolate = new Snack("Chocolate", 1.25, 200);
    Product* p_juice = new Beverage("Orange Juice", 1.75, 250);

    machine.addProduct(p_chocolate);
    machine.addProduct(p_juice);

    machine.displayInventory();

    cout << "==========Part 8==========" << endl;
    machine.insertCoin(2.00);
    machine.dispense("Chocolate");
    machine.insertCoin(2.00);
    machine.dispense("Orange Juice");

    machine.displayInventory();

    return 0;
}
```

Results (Console Output):

```
==========Part 1==========
[Constructor] Constructing VendingMachine
[Constructor] Constructing State: No Coin
[Constructor] Constructing State: Has Coin
[Constructor] Constructing State: Sold Out
Action: Initialization | Current State: No Coin | Coin Value: 0
[Constructor] Product created: Cola (ID: 1, Price: $1.5)
Beverage created: Cola (330 calories)
[Constructor] Product created: Chips (ID: 2, Price: $1)
Snack created: Chips (150 calories)
[Constructor] Product created: Water (ID: 3, Price: $1)
Beverage created: Water (0 calories)
==========Part 2==========
Current Inventory:
Beverage: Cola (ID: 1, Price: $1.5, Calories: 330 calories)
Snack: Chips (ID: 2, Price: $1, Calories: 150 calories)
Beverage: Water (ID: 3, Price: $1, Calories: 0 calories)
Total items: 3
==========Part 3==========
Action: State Changed | Current State: Has Coin | Coin Value: 1
Action: Insert Coin | Current State: Has Coin | Coin Value: 1
Action: Insert Coin | Current State: Has Coin | Coin Value: 1.5
Action: State Changed | Current State: No Coin | Coin Value: 0
[Destructor] Product destroyed: Cola (ID: 1)
Action: Dispense | Current State: No Coin | Coin Value: 0
==========Part 4==========
Action: State Changed | Current State: Has Coin | Coin Value: 1
Action: Insert Coin | Current State: Has Coin | Coin Value: 1
Action: State Changed | Current State: No Coin | Coin Value: 0
[Destructor] Product destroyed: Water (ID: 3)
Action: Dispense | Current State: No Coin | Coin Value: 0
==========Part 5==========
Action: State Changed | Current State: Has Coin | Coin Value: 0.5
Action: Insert Coin | Current State: Has Coin | Coin Value: 0.5
Insufficient funds. Please insert more coins.
Current balance: $0.5, Required: $1
```

```
Action: Dispense | Current State: Has Coin | Coin Value: 0.5
Action: Insert Coin | Current State: Has Coin | Coin Value: 1
Action: State Changed | Current State: Sold Out | Coin Value: 0
[Destructor] Product destroyed: Chips (ID: 2)
Action: Dispense | Current State: Sold Out | Coin Value: 0
==========Part 6==========
SOLD OUT: No additional coin accepted
Action: Insert Coin | Current State: Sold Out | Coin Value: 0
No product to dispense
Action: Dispense | Current State: Sold Out | Coin Value: 0
Current Inventory:
Total items: 0
==========Part 7==========
[Constructor] Product created: Chocolate (ID: 4, Price: $1.25)
Snack created: Chocolate (200 calories)
[Constructor] Product created: Orange Juice (ID: 5, Price: $1.75)
Beverage created: Orange Juice (250 calories)
Action: State Changed | Current State: No Coin | Coin Value: 0
Current Inventory:
Snack: Chocolate (ID: 4, Price: $1.25, Calories: 200 calories)
Beverage: Orange Juice (ID: 5, Price: $1.75, Calories: 250 calories)
Total items: 2
==========Part 8==========
Action: State Changed | Current State: Has Coin | Coin Value: 2
Action: Insert Coin | Current State: Has Coin | Coin Value: 2
Change returned: $0.75
Action: State Changed | Current State: No Coin | Coin Value: 0
[Destructor] Product destroyed: Chocolate (ID: 4)
Action: Dispense | Current State: No Coin | Coin Value: 0
Action: State Changed | Current State: Has Coin | Coin Value: 2
Action: Insert Coin | Current State: Has Coin | Coin Value: 2
Change returned: $0.25
Action: State Changed | Current State: Sold Out | Coin Value: 0
[Destructor] Product destroyed: Orange Juice (ID: 5)
Action: Dispense | Current State: Sold Out | Coin Value: 0
Current Inventory:
Total items: 0
[Destructor] Destructing VendingMachine
[Destructor] Destructing State: No Coin
[Destructor] Destructing State: Has Coin
[Destructor] Destructing State: Sold Out
```

Instructions for the assignment

1. Your goal is to implement all the member functions of the above classes according to the given explanation. For those functions that you already implemented in the previous assignment, you can copy-paste it.

2. You can check if your code works correctly using the sample main routine. However, your code will be graded using different test routines.

3. The required console output format can be found in the sample result. Try to print out only required information according to the given format, even though printing additional information is ok.

4. You cannot modify any access control specifiers (e.g., public, protected, private).

5. If your code fails to compile, you will get a base point; for this reason, make sure your code compliable and produce partial results that would give you partial credits.

6. You should implement this program in a single cpp file. (Do not submit additional cpp or header files)

7. Ensure that your console output matches the correct answer exactly. We will compare your output with the correct output during grading, so even a single character mismatch could result in an incorrect assessment. You can use Python or other tools to verify the accuracy of your console output.

8. You do not need to implement anything that is not specified in the instructions of this document. For example, you do not need to handle exceptions for situations where the number of products exceeds the inventory's max limit and you can implement HasCoinState::ejectCoin() function in any kind of console format.

Submission Instruction

- Submit hw3_1_YourName_YourStudentID.cpp". For example, "hw3_1_Tom_200012345.cpp"
- The source code should be complied and executed without any error.

Part 2. Written Assignment (5 pt)

Draw an UML class diagram of Part 1 Programming Assignment

Submission Instruction

- You can use any drawing tools, such as powerpoint, VISIO, hand-drawing on a paper.
- Submit an image file of the class diagram (e.g., jpg)
- The file name is hw3_2_YourName_YourStudentID.jpg (or some other image type)