CS 159: Programming Fundamentals
James Madison University, Spring 2017 Semester

PA3: Violet's Vending Venture (Version 3.0)

Due Dates

PA3-A: Wednesday, Feb. 22 at 11:00 pm

PA3-A is a Canvas online readiness quiz that must get 100% by this deadline or no credit will be given. No late submissions will receive credit.

PA3-B: Friday, March 3 at 11:00 pm

PA3-B is a WebCat submission. See submission details below for the exact files expected. After 10 submissions, 1 point will be deducted for every 2 submissions.

- -25% before Saturday, March 4, 11:00 pm
- -50% before Sunday, March 5, 11:00 pm
- Not accepted after Sunday, March 5, 11:00 pm

Honor Code

This assignment must be completed individually. Your submission must conform to the JMU Honor Code. Authorized help is limited to general discussion on Piazza, the lab assistants assigned to CS 139/149/159, and the instructor. Copying work from another student or the Internet is an honor code violation and will be grounds for a reduced or failing grade in the course. Helping someone by looking at their code is also an honor code offense.

Objectives

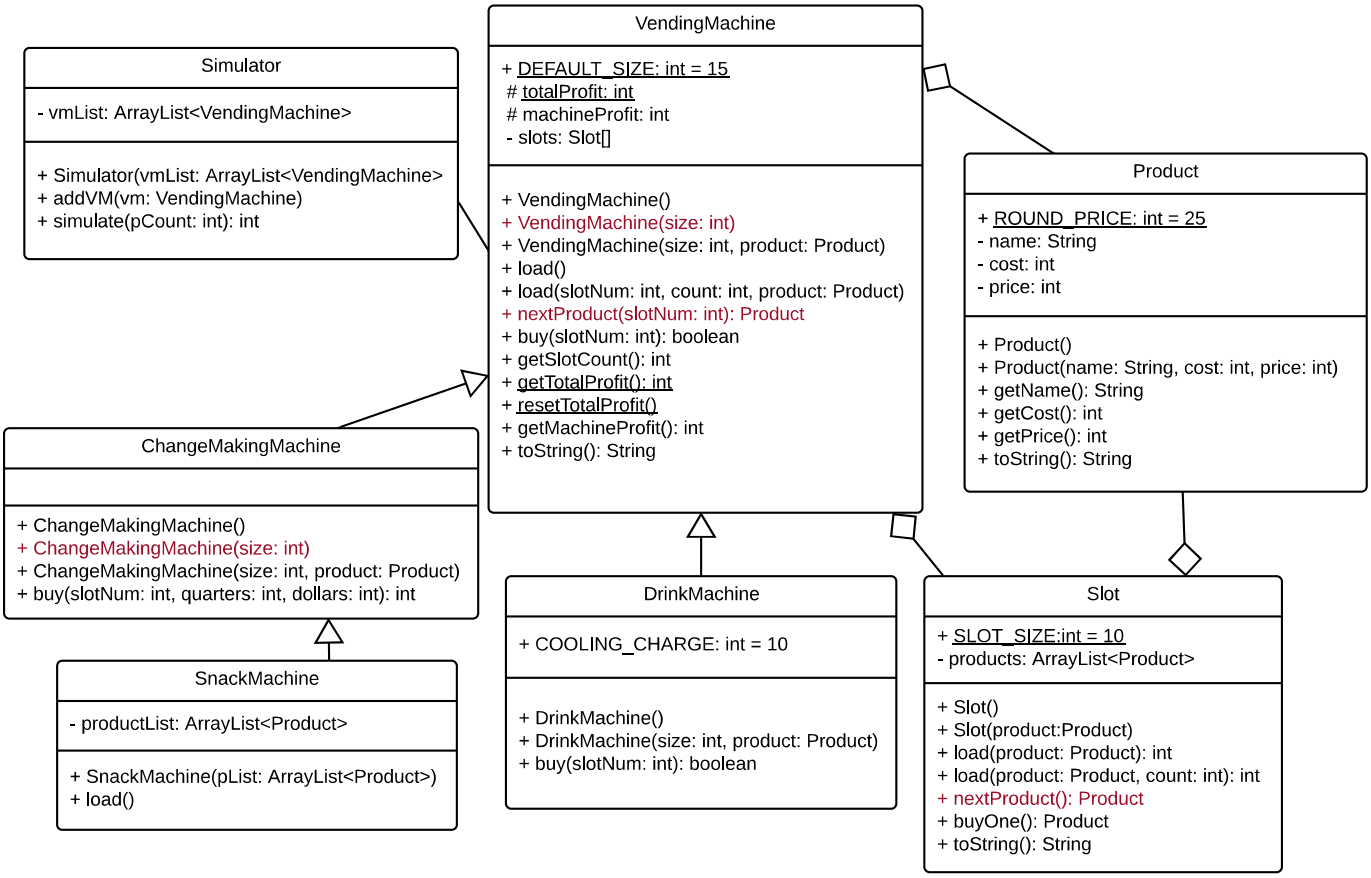
- Implement a UML diagram that uses inheritance.
- Correctly implement examples of overriding and overloading.
- Correctly predict via simulation the interaction of objects from an inheritance hierarchy.
- Implement a program that makes use of ArrayLists.

Introduction

Your friend Violet and her family are considering starting a vending machine business. To make sure it will be profitable, they have decided to keep it simple and plan! The planning part is where you get involved. She wants you to create a program that will allow her to simulate some of the possibilities of their business decisions. The goal is to have a program that represents the different kinds of vending machines they currently intend to operate with the possibility of adding different machine types in the future. The program will simulate the machines and model the profit of each machine and the whole business (the sum of the profit of

all the machines). You both realize that this means a program that uses inheritance to represent different kinds of vending machines which will make the code simpler, more flexible, and easier to maintain.

Together you decide that you will largely model profit. First to consider is the individual machine profit so that you can tell if individual machine locations are profitable. Second is the total profit of the whole business. Violet also decides that she wants to model a basic machine with only one kind of product that doesn't require anything special to operate first. But, she will consider two other kinds of machines that have a bit more sophistication in their functionality. Specifically she will want to add a drink machine, where the whole machine is still filled with the same product, but now each product will have an added cost charged to it for the refrigeration. Both the basic machine and the drink machine will require exact change. The last machine she wants to model is a traditional snack machine. It requires no refrigeration, but each slot in the machine will be loaded with a different snack product which could have a different cost, and it will return change when something is bought. In support of this vision, you have come up with the UML diagram below.



Product Class

The Product class models an individual item type in a vending machine. Each product has a name, cost. and price. Note that cost is the cost of the product to the vending machine company. Price is the price that the vending machine will charge the customer. Violet has said that she is unwilling to deal with anything but quarters, dollar bills, and debit cards so all prices are kept divisible by 25 cents. Costs to the vending machine company can be any penny value. All of the get methods perform the expected operation.

Instance Variables

Note cost and price are both integers. All money in the vending machine is represented as cents to enable simpler math and eliminate rounding issues.

ROUND_PRICE: int - the value in cents that all prices will be rounded to

name: String - the name of the product type

cost: int - cost is the cost to the vending machine company in cents

price: int - price is the price charged by the vending machine in cents and must be divisible by ROUND_PRICE.

- `Product()`

  The default constructor will create an instance of a product with a name of "Generic", a cost of ROUND_PRICE = 25 cents and a price of twice the ROUND_PRICE.

- `Product(String name, int cost, int price)throws IllegalArgumentException`

  This constructor takes the name, cost, and price as parameters and sets the instance variables appropriately. Null string names or negative cost or price should throw an IllegalArgumentException. Prices should be rounded to the next ROUND_PRICE cents above the amount given if the amount given if not already divisible by ROUND_PRICE.

  Note that if the price given is not greater than the cost, the price should be the next ROUND_PRICE divisible value greater than the cost.

- `toString()`

  The `toString()` method will return a string of the instance variables of the class exactly as shown below. Assuming a name of "M&Ms", cost of $1.02, and a price of $1.25 `toString()` would return:

  ```
  Product: M&Ms Cost: 1.02 Price: 1.25.
  ```

  Note that the cost and price are kept in cents so the `toString()` method will need to transform the values into the right format.

Slot

The Slot class models a slot in the vending machine.

Instance Variables

SLOT_SIZE: int = 10 - the maximum number of products that a slot in the vending machine can hold.

products: ArrayList<Product> - models the actual products that are in the slot. Removing the one at the front models buying one of the products in the slot and all of the others are moved forward similar to an actual vending machine.

- `Slot()`

  The Slot() constructor creates an empty array list of products.

- `Slot(Product product)`

  This constructor creates a new slot that is filled with SLOT_SIZE of product.

- `load(Product product)`

  This method loads the slot with however many new products are required to make sure it is full and returns the number of products it took to fill the slot.

- `load(Product product, int count)`

  This method loads the slot with up to count new products in an attempt to fill the slot and returns the number of products it used.

- `nextProduct()`

  This method returns a reference to the next product available for purchase. If the slot is empty this method will return `null`.

- `buyOne()`

  This method simulates the purchase of one item from the perspective of the slot. That means no money is dealt with here, rather the slot is checked to make sure there is product to buy and then one product is removed from the front of the array list modeling the slot. If a product is successfully removed from the slot, it is returned, otherwise null is returned.

- `toString()`

  The `toString()` method returns a String exactly like the one below for a slot with 10 M&M products.

  ```
  SlotCount: 10 of
  Product: M&Ms Cost: 1.02 Price: 1.25.
  ```

```
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
Product: M&Ms Cost: 1.02 Price: 1.25.
```

Hint: Don't forget to make use of other `toString()` methods.

Vending Machine

The VendingMachine class is a simple vending machine. Exact change is required so it is assumed if someone is buying something they have inserted the right amount of money or have used a debit card. The get methods return the appropriate instance variable values.

Instance Variables

DEFAULT_SIZE: int = 15 - the default size for a vending machine, used primarily by the default constructor

totalProfit: int - this models the total profit for all of the vending machines together. It is the sum of the price of every product bought from all of the machines minus the sum of the cost of all the products ever put in all of the machines. Note that it is protected in the UML diagram so it is accessible to classes that inherit from this class.

machineProfit: int - this models the long term profit for this particular machine. It is the sum of the price of every product bought from this machine minus the sum of the cost of all the products ever put in this machine. Note that it is protected in the UML diagram so it is accessible to classes that inherit from this class.

slots: Slot[] - this array models the array of slots in the vending machine.

- `VendingMachine()`

  The default constructor creates a VendingMachine with DEFAULT_SIZE empty slots.

- `VendingMachine(int size)`

  Creates a VendingMachine with the indicated number of empty slots.

- `VendingMachine(int size, Product product)`

Creates a VendingMachine with size slots each full of product.

- `load()`

  Loads an empty or partially empty VendingMachine with a Generic product for testing purposes. Makes appropriate adjustments to machineProfit and totalProfit by subtracting costs from profit values.

- `load(int slotNum, int count, Product product)throws IllegalArgumentException`

  Loads the slot indicated by slotNum with product until it is full or until count is reached. Makes appropriate adjustments to machineProfit and totalProfit by subtracting costs from profit values. Throws an IllegalArgumentException if the slotNum is out of bounds, the count is less than or equal to zero, or if the product is null.

- `nextProduct(int slotNum)throws IllegalArgumentException`

  Returns a reference to the next available product in the indicated slot or `null` if the slot is empty. Throws an IllegalArgumentException if the slotNum is out of bounds.

- `buy(int slotNum)throws IllegalArgumentException`

  Models buying one item from the slot number indicated. Makes appropriate adjustments to machineProfit and totalProfit by adding the price to the profit values. Throws an IllegalArgumentException if the slotNum is out of bounds. Returns false if there is no product to buy.

- `resetTotalProfit()`

  This method resets the totalProfit static instance variable to zero. This is useful when testing to make sure that different method tests start out in a known state for the static variable so the final value can be computed without knowing the order of the test runs.

- `toString()`

  `toString()` returns a string representing the vending machine, each slot, the machineProfit and totalProfit exactly as shown below for a 2 slot machine filled with Generic product where nothing has been bought (so the profits are negative).

```
Vending Machine
SlotCount: 10 of
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
```

```
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
SlotCount: 10 of
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Product: Generic Cost: 0.25 Price: 0.50.
Total Profit: -5.00 Machine Profit: -5.00.
```

DrinkMachine

The drink machine inherits from the general VendingMachine described above. The only additions are a constant for the cooling charge and a different buy method which will affect the profit for the machine and the total profit differently than in the general vending machine. Drink machines will assess a charge for keeping the drink cold when the drink is purchased.

Instance variables

COOLING_CHARGE: int = 10 - this models the ten cent charge assessed to each drink when it is purchased to account for the refrigeration costs of the drink machine.

- `DrinkMachine()`

  Performs the same action for the DrinkMachine as VendingMachine().

- `DrinkMachine(int size, Product product)`

  Performs the same action for the DrinkMachine as VendingMachine(int size, Product product).

- `buy(slotNum: int)throws IllegalArgumentException`

Models buying one item from the slot number indicated. Throws an IllegalArgumentException if the slotNum is out-of-bounds. Makes appropriate adjustments to machineProfit and totalProfit by adding the price (Hint: use a public method) minus the COOLING_CHARGE to the profit values.

ChangeMakingMachine

The change-making machine will add the functionality of being able to pay with cash and potentially get change back. The change will just be returned as an integer value in cents, but the amount paid in will be determined by the number of quarters and dollars that are entered.

- `ChangeMakingMachine()`

  Performs the same action for the ChangeMakingMachine as VendingMachine()

- `ChangeMakingMachine(int size)`

  Performs the same action for the ChangeMakingMachine as VendingMachine(int size).

- `ChangeMakingMachine(int size, Product product)`

  Performs the same action for the ChangeMakingMachine as VendingMachine(int size, Product product)

- `buy(int slotNum, int quarters, int dollars)throws IllegalArgumentException`

  Models buying one item from the slot number indicated. Throws an IllegalArgumentException if the slotNum is out of bounds or if quarters or dollars are negative. Computes the amount of money put into the machine in quarters and dollars, returning -1 if there is not enough money to buy the product and returning the positive difference or "change" if there is any. Makes appropriate adjustments to machineProfit and totalProfit by adding the price to the profit values if the buy is successful. (Hint: Use a public method to accomplish this.)

SnackMachine

The snack machine will inherit from the ChangeMakingMachine. The difference is it will have an additional instance variable of an array list of products which will indicate the type of product each slot should contain and it's load method will fill

each slot completely with the particular product the slot contains, making appropriate adjustments to the profit tallies.

Instance Variables

productList: ArrayList<Product> - contains the list of products for each slot in the SnackMachine. The first position in the productList corresponds to the product in the first slot in the slots array.

- `SnackMachine(ArrayList<Product> pList)`

  This constructor initializes the product list of the snack machine and creates a new snack machine where each slot is full of the product indicated in the matching position in the product list. The size of the snack machine is just the length of the product list.

- `load()`

  This load method completely fills each slot in the snack machine with the appropriate product. As a slot is filled, the total cost of the items is subtracted from the profit tallies.

Simulator

The simulator will provide a means of simulating a small business of vending machines. The vending machines of the business are stored in an array list. Vending machines can be added to the list using the provided method to simulate the growth of a business. Simulation of the business "running" and selling product is done by simulating a specific number of products being bought from every slot of every vending machine in the business and returning the totalProfit of all of the vending machines in cents.

Instance Variables

vmList: ArrayList<VendingMachine> - models the list of vending machines owned by the company

- `Simulator(ArrayList<VendingMachine> vmList)` - instantiates a simulator

- `addVM(VendingMachine vm)` - adds the vending machine indicated by vm to the end of the list of vending machines owned by the company.

- `simulate(int pCount)` - simulates buying pCount products from every slot of every vending machine owned by the company. Returns the totalProfit of all of the vending machines.

Submission

Submission for this assignment is divided into two parts that should be completed in order.

PA3-A: Readiness Quiz

In order to complete Part A you should first carefully read the project specification. Once you feel confident that you have a good grasp of the project requirements, log into Canvas and complete the Part A quiz. YOU MUST ANSWER ALL QUESTIONS CORRECTLY TO GET ANY CREDIT FOR THIS PART. You may take the quiz as many times as necessary.

PA3-B: Classes defined in UML and Test Classes for all but the Simulator class

For this part you must submit your `Product.java, ProductTest.java, Slot.java, SlotTest.java, VendingMachine.java, VendingMachineTest.java, DrinkMachine.java, DrinkMachineTest.java, ChangeMachine.java, ChangeMachineTest.java, SnackMachine.java, SnackMachineTest.java, and Simulator.java` files to WebCat.

Code submitted for Part B must conform to the course style guide. You do not need to submit
driver code and checkstyle will not be run on your test classes. Your test classes should provide 100% branch coverage.

You will receive at most 50% of the possible points for part B if your code fails any of the instructor unit tests.

Grading

Part A	10%
Part B Web-CAT Correctness/Testing	60%
Part B Web-CAT Checkstyle	10%
Part B Instructor grading based on style and code quality	20%

Submission penalties

This assignment will include a penalty for excessive Web-CAT submissions. You are allowed ten submissions with no penalty. There will be a deduction of 1 point for every two additional submissions beyond ten.