Project #1: Vending Machine

In this project you will model a vending machine.

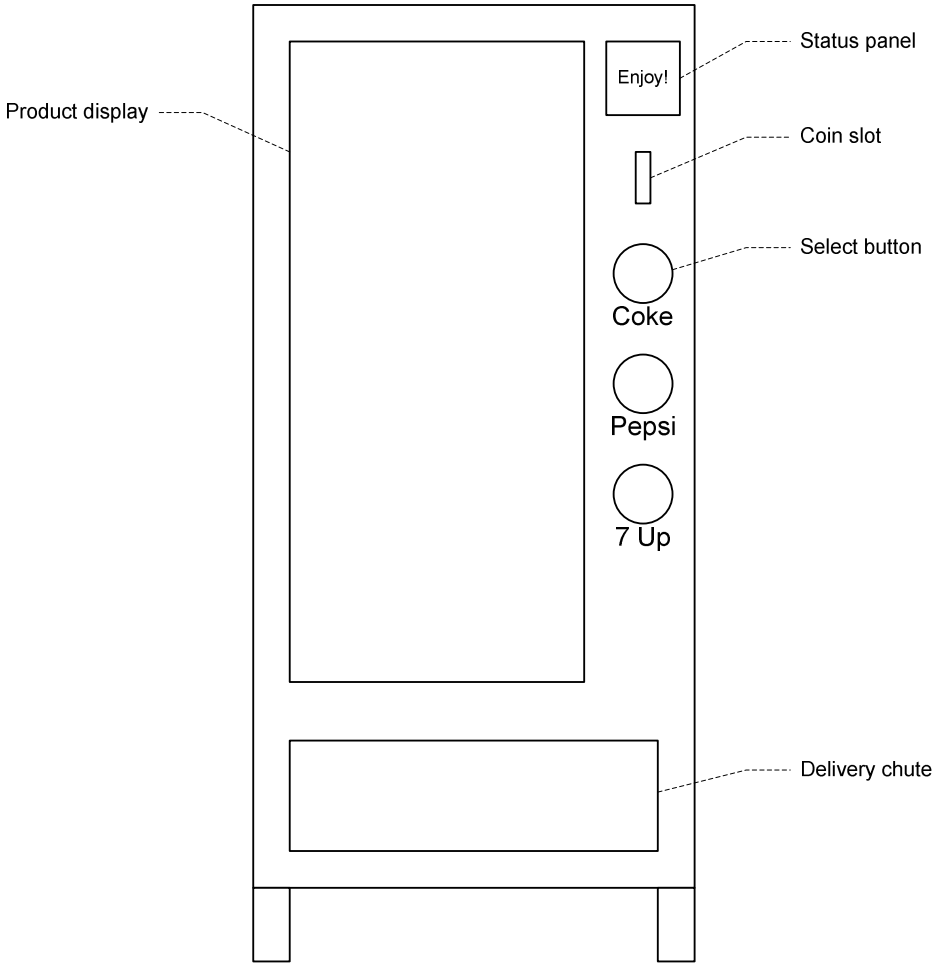

Figure 1. Vending machine

Principals of Operation

You can imagine the external interface of the vending machine including a product display area allowing customers to see the available products, a status panel that displays messages when users take actions (e.g. to tell the user she must insert more money if not enough money has been inserted for a purchase), a coin slot for inserting money, a set of buttons for selecting a product to purchase, and a delivery chute from which customers may retrieve a purchased product.

Internally the vending machine contains a set of product racks. Each product rack holds a set of products all of the same type.

Two types of individuals may use a vending machine, a customer and a service professional. The customer can insert coins into the vending machine, press buttons to

purchase products (sodas), and retrieve products from the vending machine's delivery chute one at a time. The service professional can add products to the vending machine, request the number of products of a certain type currently in the machine, request the current total balance of coins, and empty the till of all of the coins.

The following UML class diagram shows the classes and operations that will be used to model the vending machine.

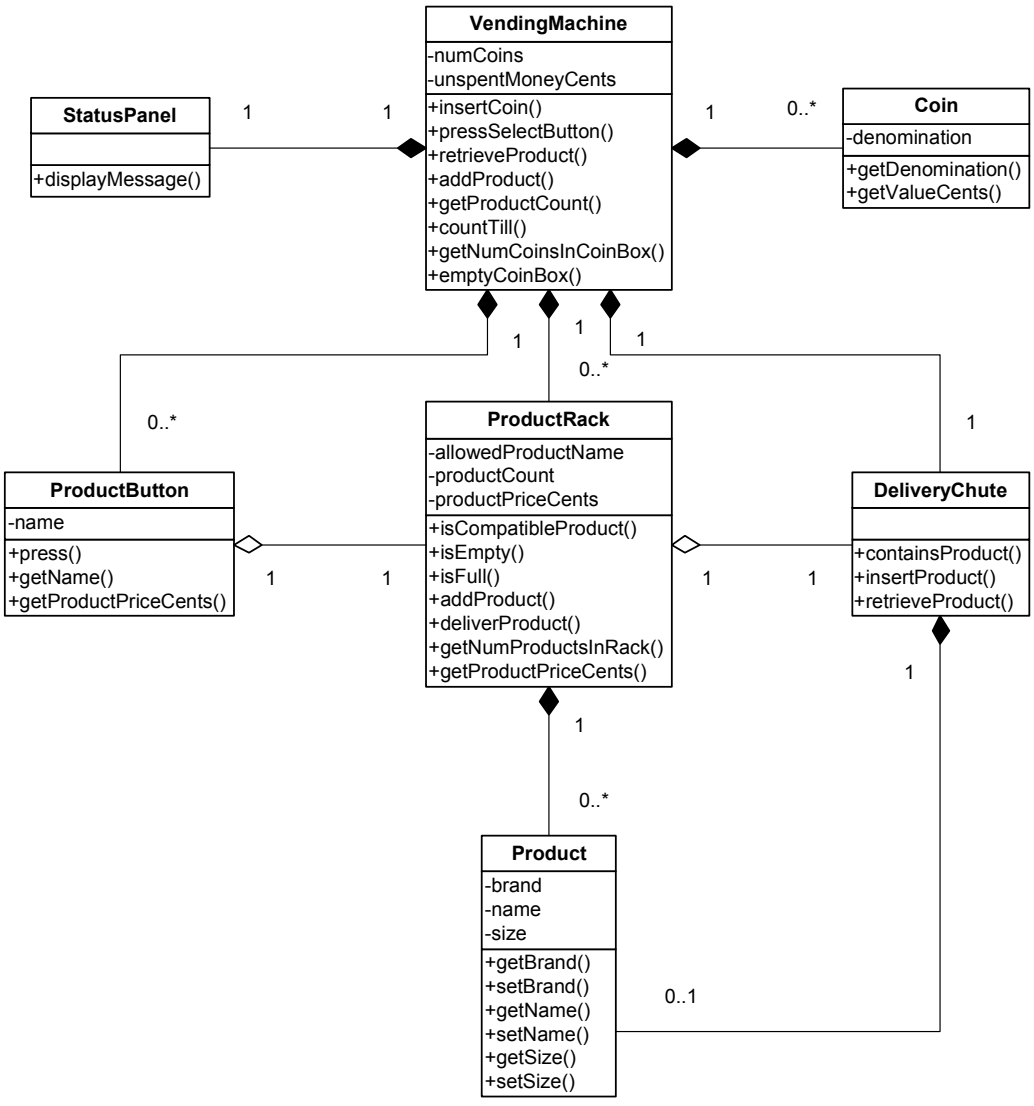

Figure 2. UML class diagram for vending machine.

Requirements

The following requirements must be met by the vending machine implementation (more details are provided in the provided source code).

- Any operation that fails (such as inserting an invalid coin into the machine) should cause an appropriate message to be displayed on the status panel.

- Each product rack is associated with a specific type of product. If an attempt is made to add a product of a type with no matching product rack then the attempt should fail.
- Each product rack is limited to holding a maximum number of products. An attempt to add a product to an already full product rack should fail.
- The vending machine accepts the following types of coins: dollars, half-dollars, quarters, dimes, nickels, and pennies. Any other type of coin (e.g. wooden nickel) inserted into the machine should be rejected.
- An attempt to purchase a product when the product rack containing that product is empty should fail.
- The delivery chute may only contain a single product at a time. If an attempt is made to purchase a product when another product is already in the delivery chute then the purchase should fail.
- Pushing a button on the vending machine entails passing the button label to a method. If no button with that label exists in the vending machine the operation should fail.
- Attempting to purchase a product when insufficient coins have been inserted into the vending machine should fail.

Project Files

Files you must implement

These files contain the class member function definitions for the classes required by the vending machine. **You must complete the implementations of the member functions in these files:**

- Coin.cpp
- DeliveryChute.cpp
- Product.cpp
- ProductButton.cpp
- ProductRack.cpp
- VendingMachine.cpp

Files you must not change

The following files have already been implemented and must not be changed. I will use my own original versions of these files when grading. **These files must not be modified:**

- Vending Machine Header Files
  - Coin.h
  - DeliveryChute.h
  - Product.h
  - ProductButton.h
  - ProductRack.h
  - StatusPanel.h
  - VendingMachine.h
- Status Panel Implementation File - *This implementation has been provided for you because we have not yet covered ostreams*

    - StatusPanel.cpp –
- Test Framework – *Generic test framework for running unit tests (will be reused for project #2)*
    - TestFramework.cpp
    - TestFramework.h
- Unit Tests – *These tests test the logic of the functions you will be implementing*
    - main.cpp
    - UnitTest.cpp
    - UnitTest.h

Work required

You will need to provide an implementation for each of the member functions in the "Files you must implement" section.

Grading

You will be graded based on how your solution performs against the unit tests and how few memory leaks your program has.

You can earn a number of points on this project equal to the total number of unit tests present in UnitTest.cpp. You will lose 1 point for each unit test your program fails. You will lose 1 point for each unique memory leak your program exhibits. **If your program fails to compile you will receive 0 points.**

Turning in the project

Submit a single archive file (e.g. .zip, .tar) containing only the files listed under the "Files you must implement" section. *I will not use any other files you send; I will be using my original versions of all other files.*

Note: You may submit your project early if you would like. I will let you know what grade your project would receive. You may then make corrections and resubmit. You may repeat this as many times as desired up until the due date.
