



## CHAPTER 9

# OO Design with CRC Cards



### A sample user story

Suppose that the customer of this project is a manufacturer of vending machines for soft drinks. They request us to develop software to control their vending machines. Suppose that the customer describes the following user story:

**Name: Sell soft drink**

Events:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.
3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
5. The vending machine sells a can of soft drink to him.
6. The vending machine returns the changes to him.

In order to implement this user story, what classes do we need? Let's implement each step in the user story, as shown below.

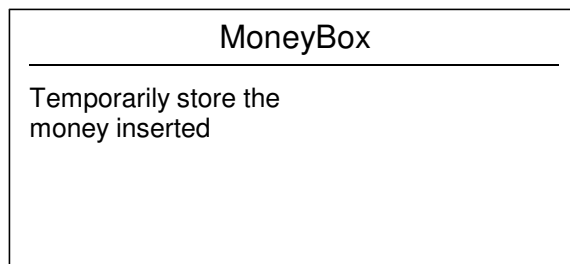
### Design for a user story

In order to implement steps 1 and 2 in the user story, we need to find out the behaviors that the system needs to perform:

1. A user inserts some money.
  - Behavior 1: Note that someone has inserted some money.

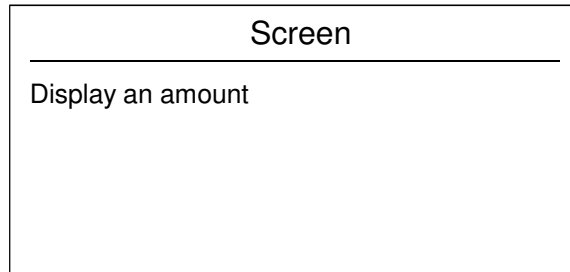
2. The vending machine displays how much money he has inserted so far.
  - Behavior 2: Find out how much money the user has inserted so far.
  - Behavior 3: Display it.
3. ...
4. ...
5. ...
6. ...

Who will perform these behaviors? Suppose that the money box is represented by a MoneyBox class, that it is alerted whenever someone inserts some money (behavior 1) and that it keeps track of how much money the user has inserted (behavior 2) (we write the following on a 4 inch by 6 inch index card by hand):

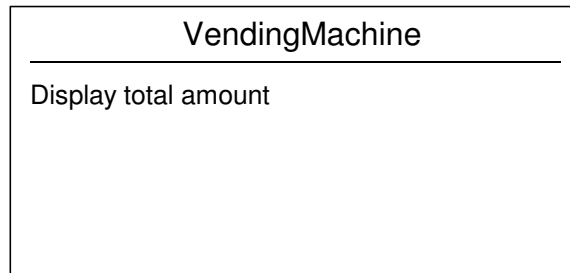


The phrase "temporarily store the money inserted" written on the card is a "responsibility". In order to implement this responsibility, MoneyBox may need quite a few attributes and methods such as a balance attribute plus methods like putMoney, takeMoney, returnMoney, getAmount and etc. At the moment, we hope to think on a high level, therefore we think in terms of responsibilities instead of attributes and methods.

OK, let run it again. The user inserts some money. MoneyBox is alerted. It knows how much money has been inserted. However, who is going to display the total amount (behavior 3)? Suppose that there is a Screen class (make a new card):

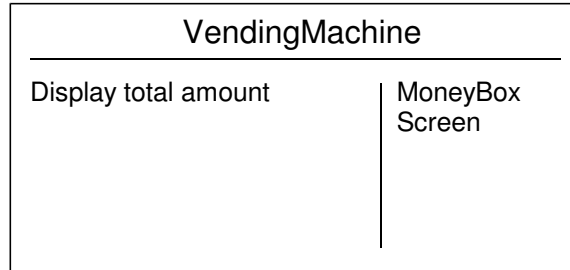


Who is responsible for sending the amount from MoneyBox to Screen? (put one hand on the MoneyBox card and the other on the Screen card) How about creating a VendingMachine class (make a new card):



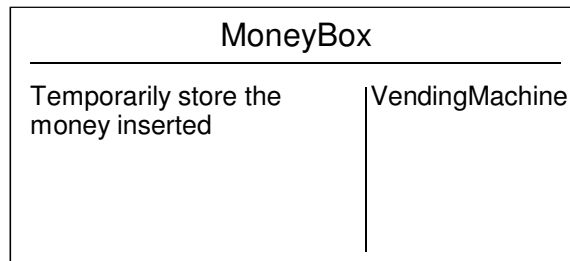
A user inserts some money. MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine). VendingMachine gets the total amount from MoneyBox (move VendingMachine to MoneyBox), then asks Screen display the total amount (move VendingMachine to Screen).

Suppose that we hope to emphasize that VendingMachine needs to work with MoneyBox and Screen, we can update the VendingMachine card as:



The words "MoneyBox" and "Screen" on the right side of the card state: VendingMachine needs to collaborate with MoneyBox and Screen, i.e., MoneyBox and Screen are both "collaborators" of VendingMachine.

Because MoneyBox needs to notify VendingMachine whenever someone inserts some money, VendingMachine is also a collaborator of MoneyBox. Therefore, if we think there is a need to emphasize this, we can change MoneyBox to:



Here we assume that there is no such a need.

Because on these cards we write the name of a class, its responsibilities and the collaboration relationships, we call these cards "CRC cards". The letters "CRC" stand for Class, Responsibility and Collaboration respectively.

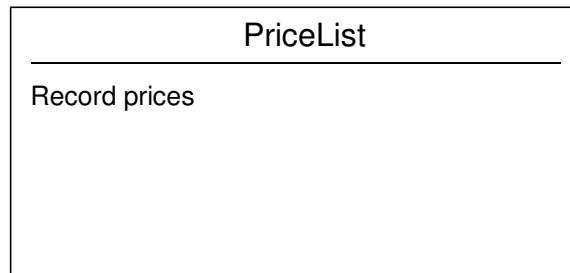
Now, let's find out the behaviors needed to be performed by the system in order to implement step 3 in the user story:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

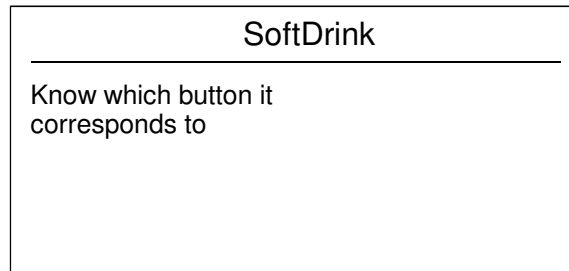
3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
  - Behavior 1: Note that someone has inserted some money.
  - Behavior 2: Find out how much money the user has inserted so far.
  - Behavior 3: Determine the price of each type of soft drink.
  - Behavior 4: Determine which button corresponds to which type of soft drink.
  - Behavior 5: If the total amount exceeds the price of a type of soft drink, light up the corresponding button.
4. ...
5. ...
6. ...

Who is going to perform these behaviors? Suppose that MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine) that someone has inserted some money (behavior 1). VendingMachine gets the total amount from MoneyBox (behavior 2), then find out the price of each type of soft drink (behavior 3). From where does it get the prices? It may be a PriceList object (make a new card):

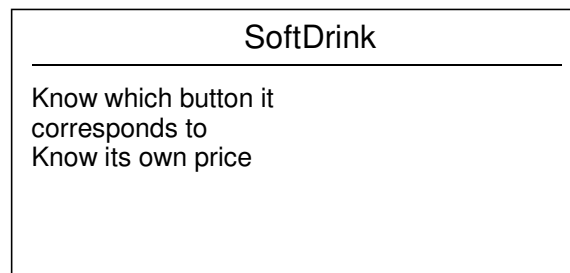


After VendingMachine gets the prices (move VendingMachine to PriceList), it needs to determine to which button each type of soft drink corresponds. How about creating a SoftDrink (make a new card) object to represent a type of soft drink and let it record to which button it corresponds (behavior 4):

As we have SoftDrink now, why not let it record its own price? Therefore we no longer need

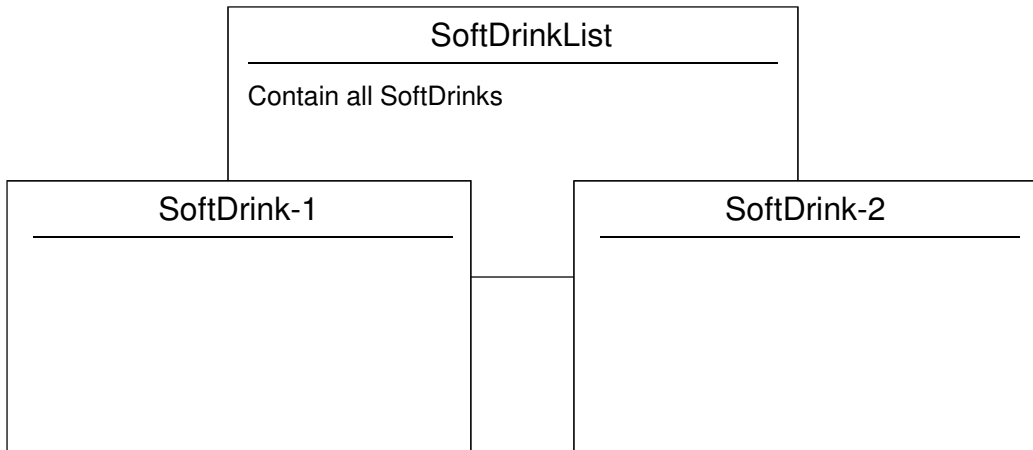


PriceList (tear it up and throw it into the trash bin). However, we still need an object to contain all the SoftDrinks. Let's call it SoftDrinkList (make a new card):

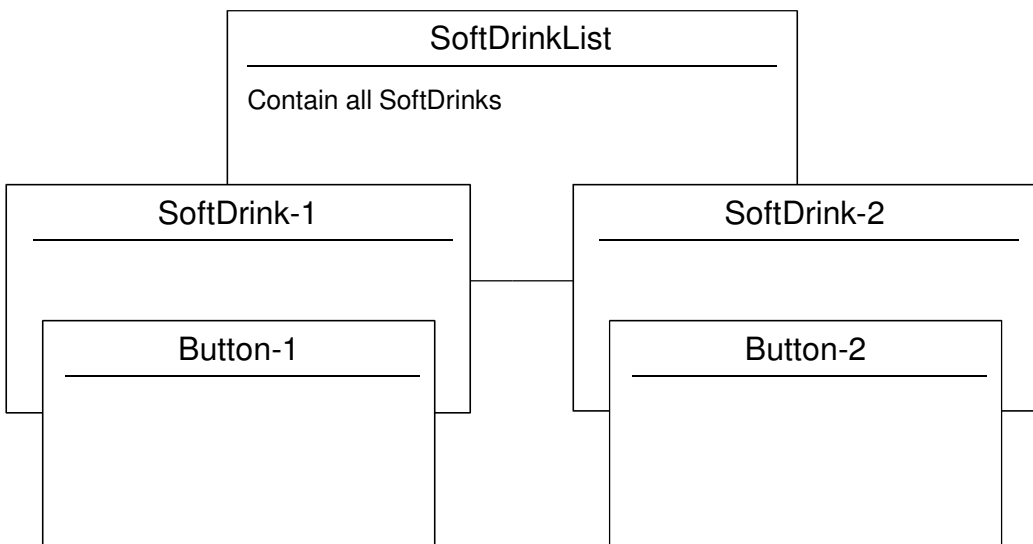


OK, let run it again. MoneyBox notifies VendingMachine (move MoneyBox to VendingMachine) that someone has inserted some money (behavior 1). VendingMachine gets the total amount from MoneyBox (behavior 2), then find out all types of soft drinks (move VendingMachine to SoftDrinkList). Suppose that there are two types of soft drinks (rename SoftDrink as SoftDrink-1, make another card named SoftDrink-2, then put them under SoftDrinkList):

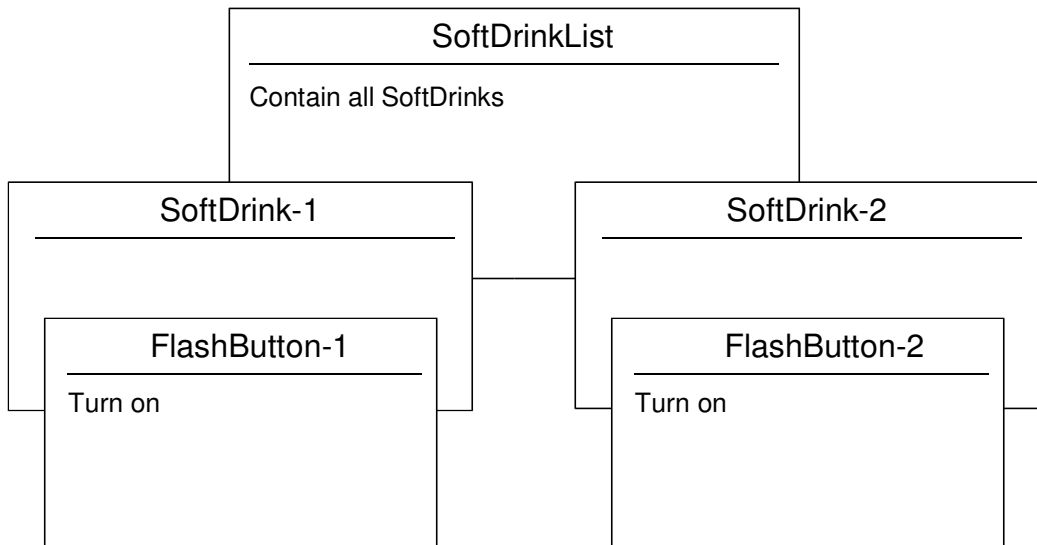
Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)



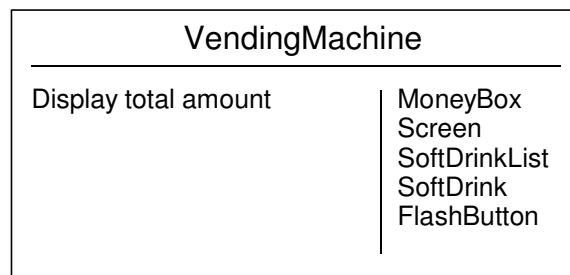
VendingMachine asks SoftDrink-1 for its price (move VendingMachine to SoftDrink-1). Suppose that the total amount is not enough. VendingMachine asks SoftDrink-2 for its price (move VendingMachine to SoftDrink-2). Suppose that the total amount is enough. VendingMachine asks SoftDrink-2 to which button it corresponds (move VendingMachine to SoftDrink-2). Suppose that it corresponds to a certain Button (make a card for the button and put it along with SoftDrink-2). Also make one for SoftDrink-1):



Finally, VendingMachine asks Button-2 to light itself up. As a Button is responsible for lighting itself up, how about renaming it to FlashButton:



OK, we have finished the behaviors needed by step 3. If required, we may update VendingMachine like:



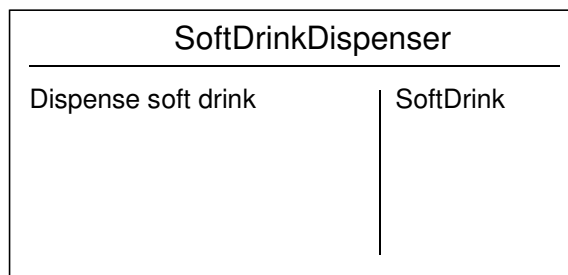
Now, let's find out the behaviors needed to be performed by the system in order to implement steps 4 to 6 in the user story:

1. A user inserts some money.
2. The vending machine displays how much money he has inserted so far.

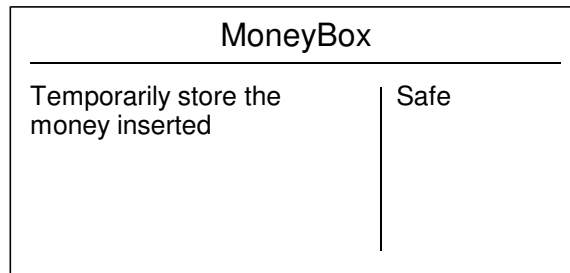
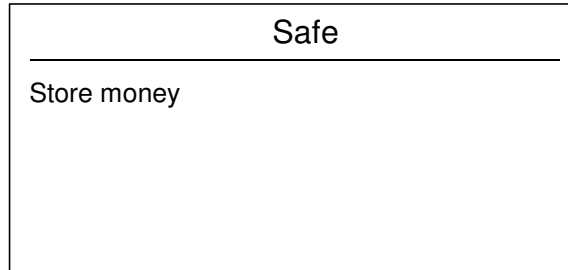
Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agileskills.org](http://www.agileskills.org)

3. If the money is enough to buy a certain type of soft drink, the vending machine lights up the button representing that type of soft drink.
4. The user presses a certain lit-up button.
  - Behavior 1: Note that someone has pressed a button.
5. The vending machine sells a can of soft drink to him.
  - Behavior 2: Check to make sure the button is lit-up.
  - Behavior 3: Find out the type of soft drink corresponding to that button.
  - Behavior 4: Dispense a can of soft drink of this type.
  - Behavior 5: Turn off the button.
6. The vending machine returns the changes to him.
  - Behavior 6: Return the changes.

The user presses a button e.g. FlashButton-2 (put left hand on FlashButton-2). FlashButton-2 checks if it is lit up. If yes, it notifies VendingMachine (put right hand on VendingMachine. Move FlashButton-2 to VendingMachine). VendingMachine asks SoftDrink-1 if this is its button (move VendingMachine to SoftDrink-1). It says no. VendingMachine asks SoftDrink-2 if this is its button (move VendingMachine to SoftDrink-2). It says yes. Therefore, VendingMachine dispenses a can of soft drink belong to type SoftDrink-2. Who is responsible for this action? It may be a SoftDrinkDispenser (make a new card):



Then VendingMachine turns off FlashButton-2 (move VendingMachine to FlashButton-2), asks SoftDrink-2 for its price (move it to SoftDrink-2), asks MoneyBox for the total amount the user has inserted (move it to MoneyBox), tells MoneyBox to put the money into the safe (make a new card to represent the safe and move MoneyBox to it):



Then VendingMachine calculates the changes and tells the Safe to return the changes (move VendingMachine to Safe).

## Typical usage of CRC cards

Why use CRC cards instead of Word or the advanced CASE tools?

1. The space on a card for writing is very limited, prohibiting us from writing too many responsibilities. If there are many responsibilities (e.g., more than 4), check if we can write in a more abstract way, merging several responsibilities into one. If no, consider creating a new class to share its responsibilities.
2. CRC cards are mainly used for exploring or discussing various design alternatives. A design doesn't work? Simply throw the cards away. In addition, once the design is done, we can throw all the cards away. Their purpose is not for documentation.
3. We can move the CRC cards back and forth or put the related cards together.

Licensed for viewing only. Printing is prohibited. For hard copies, please purchase from [www.agilekills.org](http://www.agilekills.org)

CRC cards are mainly used for discussing and establishing a quick design. We should not spend a lot of time in this activity, nor should we expect the design created to be working perfectly (in most of the time it does not entirely work). When we code, we still need to continuously adjust the design. Only when we design and code together can we efficiently create good designs.

There is no formal definition for the usage of CRC cards. We don't need to care too much what to write on each card, or whether something is missing on the cards. For some people, maybe writing the class names is already enough. Some people may consider that it helps them think better if the responsibilities are also written. Some other people may hope to also write the collaborators. We don't need to care too much what it means to put one card on another, or what it means to move one card to another. As long as we keep talking while we manipulate the cards, the meaning of the actions will become very clear.

## References

- <http://c2.com/doc/oopsla89/paper.html>.
- <http://c2.com/cgi/wiki?CrcCards>.



## Chapter exercises

1. Use CRC cards to design for the user story: A user inserts his ATM card into an ATM. The system asks him for the password. He inputs the password. The system asks him to choose an operation. He chooses withdrawal. The system asks how much he would like to withdraw. He inputs the amount. The system gives him the money. The system asks him to choose an operation. He chooses quit. The system ejects the ATM card.

You must use real cards to design. You must not use Word or some other software instead. Try moving the cards in the process to help you think.