Université catholique de Louvain
École polytechnique de Louvain
Pôle d'ingénierie informatique

LINGI2251

Development methods

Assignment 2: The Dinoco GSCS

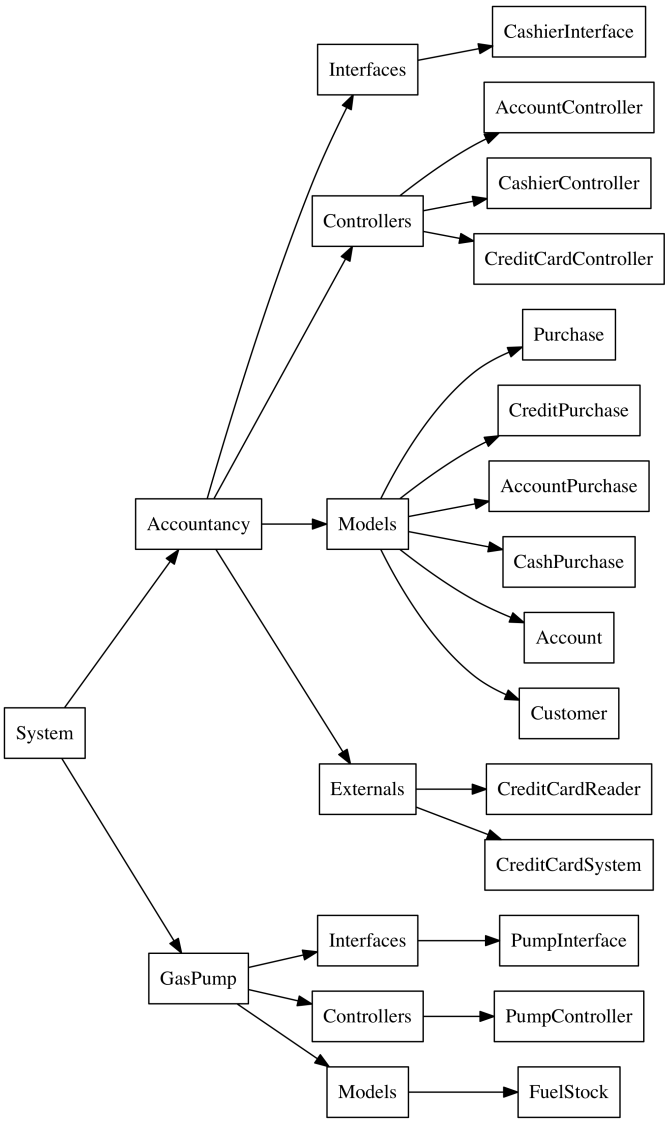
April 20, 2015

Professor
Charles Pecheur

Michael Heraly
Thibault Gerondal

1 Architectural Design

1.1 Hierarchical decomposition



1.2 Roles and interactions

Our system is composed of two parts : “Accountancy” and “Gas Pump”. The first one is related to accounting. And the second one is about the logistic of the gas pump. This hierarchy allows to easily extend the activities of **Dinoco GSCS**.

In each of these two sections, we defined three main components :

Interfaces contains components that will interact with the user (GUI, buttons).

Controllers is responsible of interactions between components.

Models contains data.

We added the component "Externals" in the section "Accountancy" for the external devices.

Descriptions of the system components

CashierInterface is responsible for displaying the graphical user interface to the cashier.

AccountController is responsible for the access to accounts informations.

CashierController is responsible for the interactions between the `CashierInterface` and the `AccountController` (e.g. open a monthly bill, pay a monthly bill, etc.). It also permits the cashier to check the fuel stock from the `PumpController`.

CreditCardController is responsible for the interactions between the `CreditCardReader` and the `CreditCardSystem`.

Purchase, CreditPurchase, AccountPurchase, CashPurchase are responsible for representing data about purchase.

Account, Customer are responsible for representing data about accounts.

CreditCardReader is responsible for reading credit card number (wrapping the existing API for the device).

CreditCardSystem is responsible for requesting payment (wrapping the existing system API).

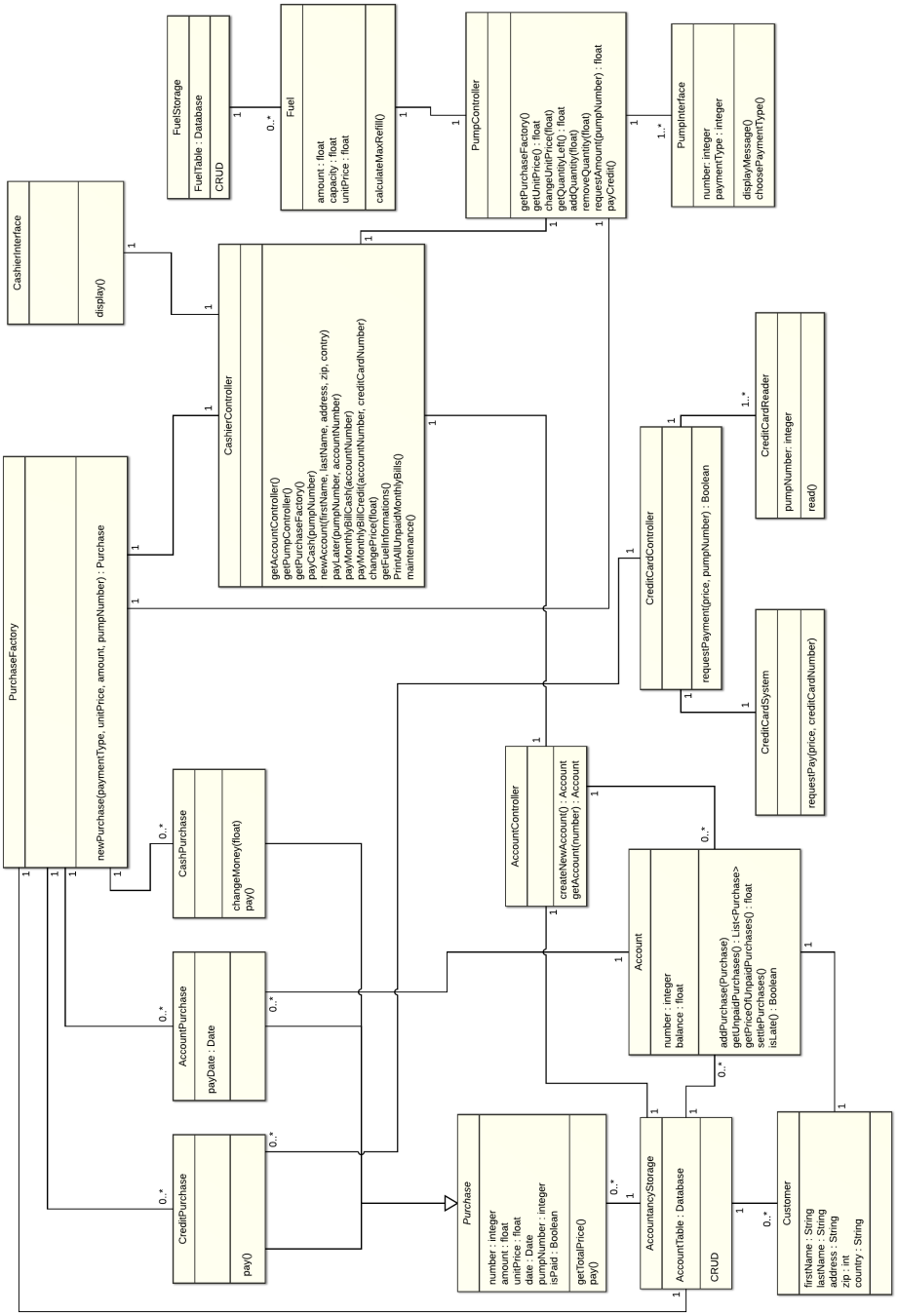
PumpInterface is responsible for displaying informations to the customer.

PumpController is responsible for the interactions between the `PumpInterface` and the Dinoco GSCS.

FuelStock is responsible for representing data about fuel.

2 Detailed design

Class diagram



Short description

CashierController

The `CashierController` is the main controller. It is the entry point of the application. It will initialize the `PumpController`, the `AccountController`, the `PurchaseController` and finally its interface (`CashierInterface`).

CashierInterface

This class is responsible of displaying the graphical user interface to the cashier.

PurchaseFactory

The `PurchaseController` is responsible of creating and saving new `Purchase`. If the purchase is a `CreditPurchase`, the controller will interact with the `CreditCardController` to directly settle the payment.

Purchase

This abstract class represents a purchase.

CreditPurchase

Specialized class for representing a purchase made via credit card.

AccountPurchase

Specialized class for representing a purchase that is not paid immediately. The purchase will be associated to a monthly bill account (`Account` class).

CashPurchase

Specialized class for representing a purchase made via cash.

Account

This class is responsible for keeping up to date the accountancy of a customer.

AccountController

This controller is responsible for creating new `Account` and getting existing `Account`.

Customer

Class representing a customer.

AccountancyStorage

This class is responsible of the persistence (to keep in a database) of classes : `Account`, `Purchase` and `Customer`.
It provides CRUD methods: `create`, `read`, `update`, and `delete`.

CreditCardController

This class will interact with the `CreditCardReader` and the `CreditCardSystem`. It will offer a simple way to request a payment by manipulating the external APIs of the credit card company. We need to know the `pumpNumber` to enable the right card reader associated to a gas pump.

CreditCardSystem

This class will wrap the external API of the credit card system.

CreditCardReader

This class will wrap the external API of the credit card reader.

PumpInterface

This class is responsible for displaying messages and listen to choices made by the user.

PumpController

This controller manages everything related to the `Fuel` stock. Since the user can pay directly by credit card at a pump, it can call the `PurchaseFactory` to handle the payment.

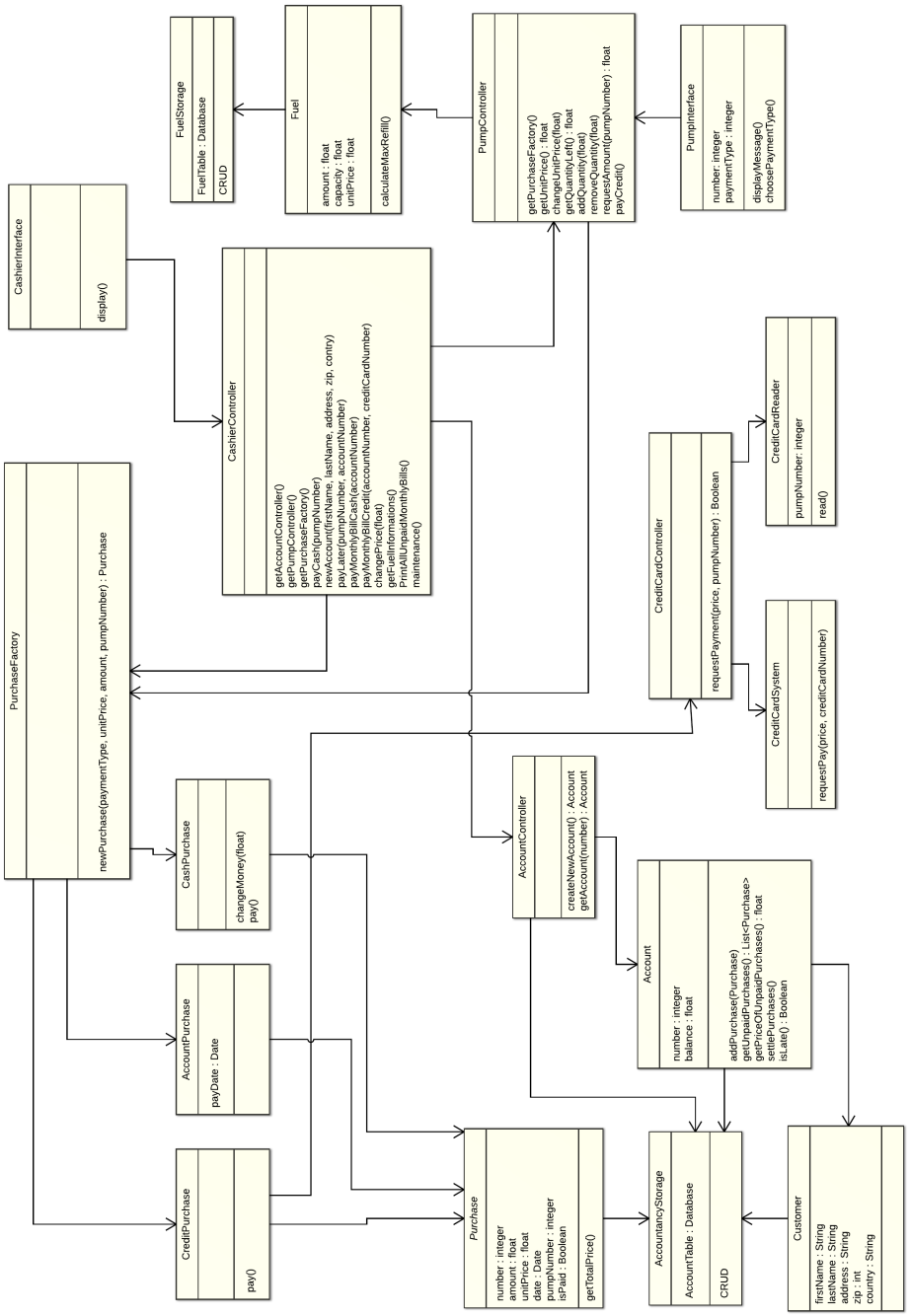
Fuel

Class for representing fuel stock.

FuelStorage

This class is responsible of the persistence (to keep in a database) of the `Fuel` stock.

USES diagram



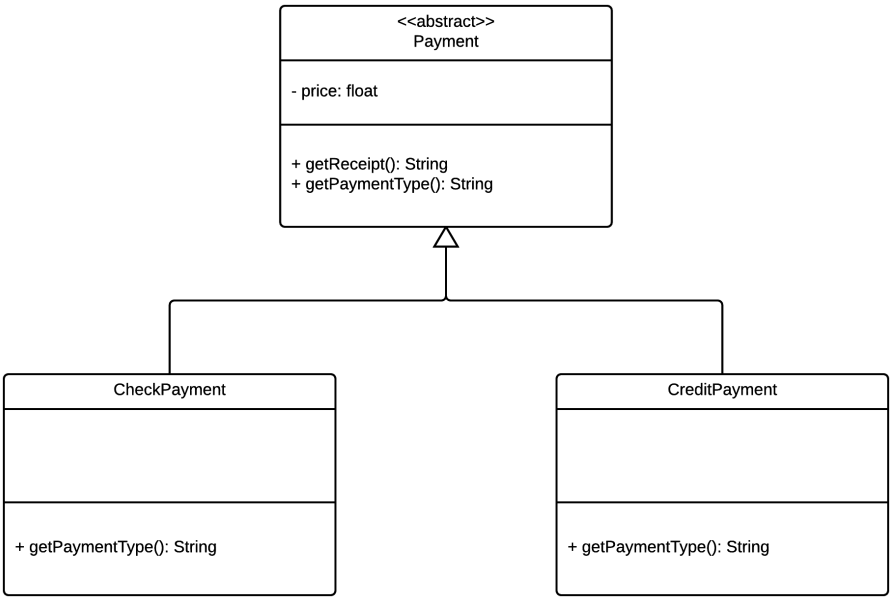
Sets of classes for an incremental development

We do not have any loop in our USES diagram. The incremental development below could be followed :

- FuelStorage, AccountancyStorage, CreditCardSystem, CreditCardReader
- Purchase, Customer, Account, CreditCardController, Fuel
- CreditPurchase, AccountPurchase, CashPurchase
- PurchaseFactory, AccountController
- PumpContoller
- CashierController, PumpInterface
- CashierInterface

Design Patterns

Template Method



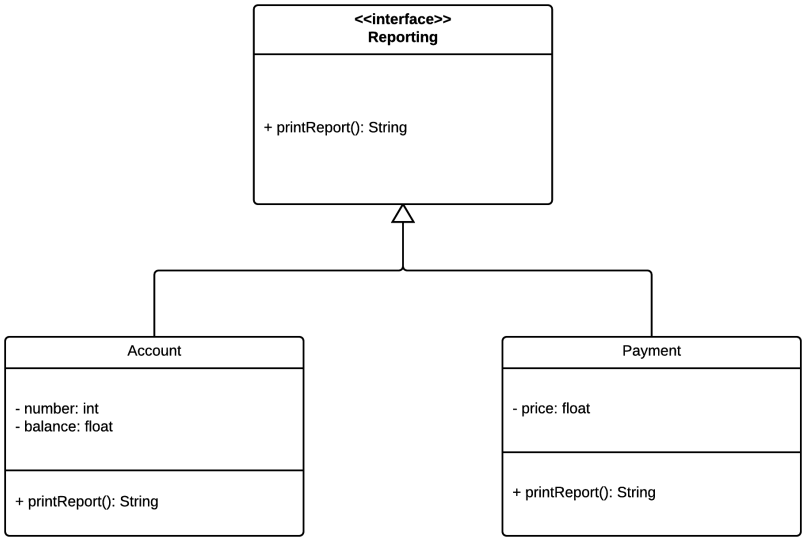
Any instance of `Payment` has a method `getReceipt()` that produces a printable receipt for the payment. This method calls the `getPaymentType()` method that subclasses of `Payment` must implement.

```
public abstract class Payment {
    public String getReceipt() {
        return "Payment by "+getPaymentType()+": "+this.price+"$";
    }

    public abstract String getPaymentType();
}
```

```
class CheckPayment extends Payment {
    // ...
    private String getPaymentType() {
        return "check";
    }
    // ...
}
```

```
class CreditPayment extends Payment {
    // ...
    private String getPaymentType() {
        return "credit card";
    }
    // ...
}
```

Strategy pattern



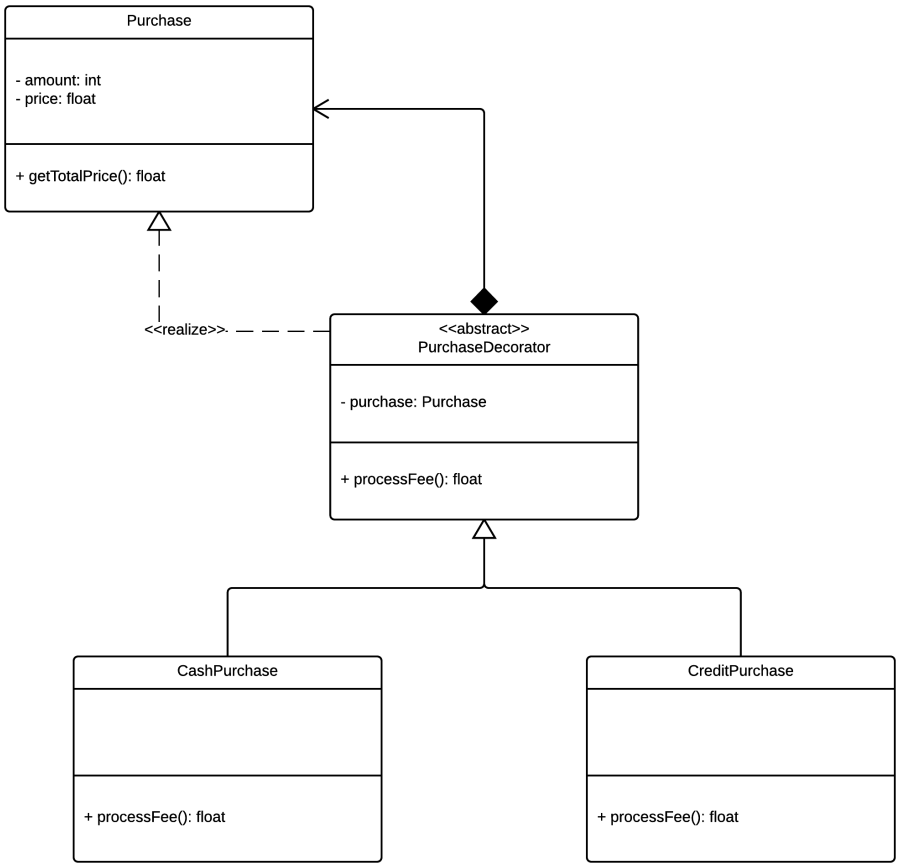
The `Reporting` interface advertises the `printReport` method. Each of the classes implementing `Reporting` must define its own `printReport` method.

```
public interface Reporting {
    public String printReport();
}
```

```
public class Account implements Reporting {
    // ...
    public String printReport () {
        String report = "";
        List<Payment> payments = getPayments();
        for (Payment p : payments) {
            report += p.printReport()+"\n";
        }
        return report;
    }
    // ...
}
```

```
public class Payment implements Reporting {
    // ...
    public String printReport () {
        String report = "Payment informations:\n"
                      +"Date: "+getDate()+"\n"
                      +"Amount: "+getTotalPrice()+"\n";
        return report;
    }
    // ...
}
```

Decorator pattern



The `PurchaseDecorator` class inherits from `Purchase`, and contains a `Purchase` object. Classes that inherit from `PurchaseDecorator` can add or remove functionalities at runtime.

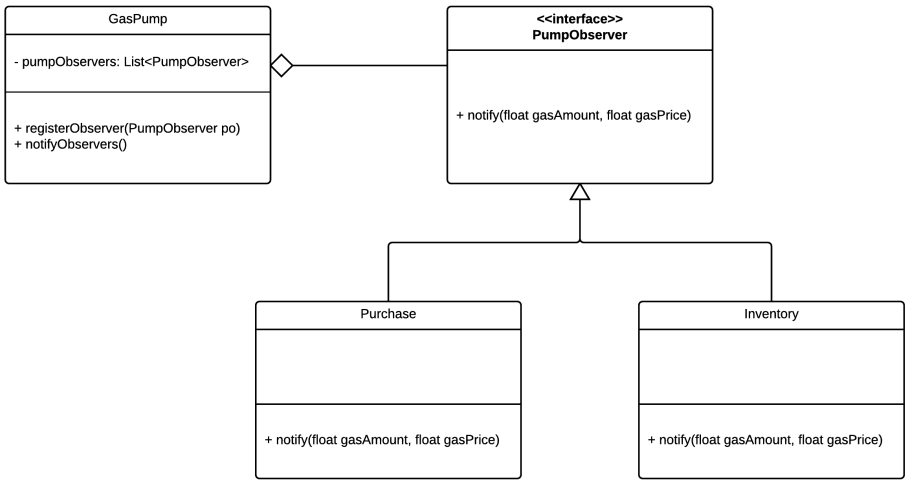
```
public class Purchase {
    // ...
}
```

```
public abstract class PurchaseDecorator extends Purchase {
    protected Purchase purchase;

    public PurchaseDecorator (Purchase p) {
        this.purchase = p;
    }

    public abstract float processFee();
}
```

```java
public class CashPurchase extends PurchaseDecorator {
    // ...
    public float processFee() {
        return getTotalPrice() + 0.50;
    }
}
```

```java
public class CreditPurchase extends PurchaseDecorator {
    // ...
    public float processFee() {
        return getTotalPrice() * 1.05;
    }
}
```

Observer Pattern



The `GasPump` has a collection of `PumpObserver`. When the customer hangs up the nozzle, the `GasPump` executes the `notifyObservers` method, which is responsible for notifying to the observers the amount and the price of gas that the customer dispensed.

```java
public class GasPump {
    private List<PumpObserver> pumpObservers;
    // ...
    public void registerObserver(PumpObserver po) {
        this.pumpObservers.add(po);
    }

    public void notifyObservers () {
        float dispensedGasAmount = getDispensedGasAmount();
        float dispensedGasPrice = getDispensedGasPrice();
        for (PumpObserver po : this.pumpObservers) {
```

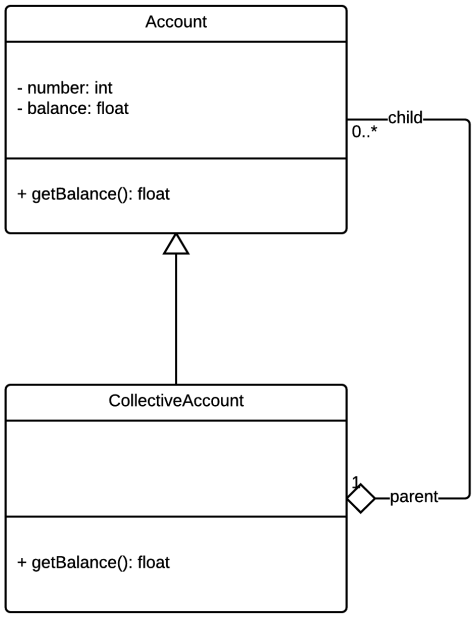
```
            po.notify(dispensedGasAmount, dispensedGasPrice);
        }
    }
}
```

```
public interface PumpObserver {
    public void notify(float gasAmount, float gasPrice);
}
```

```
public class Purchase implements PumpObserver {
    // ...
    public void notify (float gasAmount, float gasPrice) {
        // ...
    }
}
```

```
public class Inventory implements PumpObserver {
    // ...
    public void notify (float gasAmount, float gasPrice) {
        // ...
    }
}
```

Composite Pattern



The `CollectiveAccount` is at the same time an `Account`, but also an aggregation of `Accounts`. The composite pattern allows to treat composite and non-composite components uniformly.

```
public class Account {
    protected int number;
    private float balance;

    public float getBalance() {
        return this.balance;
    }
}
```

```
public class CollectiveAccount extends Account {
    List<Account> accounts;
    // ...

    public float getBalance() {
        float total = 0.0;
        for (Account a : accounts) {
            total += a.getBalance();
        }
        return total;
    }
}
```