LINGI2251
Software Engineering: Development Methods
Assignment 2

Matthieu Baerts
Olivier Nauw

April 16, 2016

Contents

1 Architectural Design **2**
1.1 Hierarchical decomposition . . . 2
1.2 Roles and interactions . . . 2

2 Detailed Design **3**
2.1 Class diagram . . . 3
2.2 Short description of each class . . . 3
2.2.1 `MainController` . . . 3
2.2.2 `CashierController` . . . 3
2.2.3 `PumpController` . . . 3
2.2.4 `GasPump` . . . 4
2.2.5 `Fuel` . . . 4
2.2.6 `FuelInventory` . . . 4
2.2.7 `Purchase` . . . 4
2.2.8 `CashPurchase` . . . 4
2.2.9 `AccountPurchase` . . . 4
2.2.10 `CreditPurchase` . . . 4
2.2.11 `CustomersController` . . . 4
2.2.12 `Customer` . . . 4
2.2.13 `Address` . . . 4
2.2.14 `Account` . . . 4
2.2.15 `Payment` . . . 5
2.2.16 `CheckPayment` . . . 5
2.2.17 `CreditPayment` . . . 5
2.2.18 `CreditTransaction` . . . 5
2.2.19 `ExternalDevicesController` . . . 5
2.3 USES diagram . . . 5

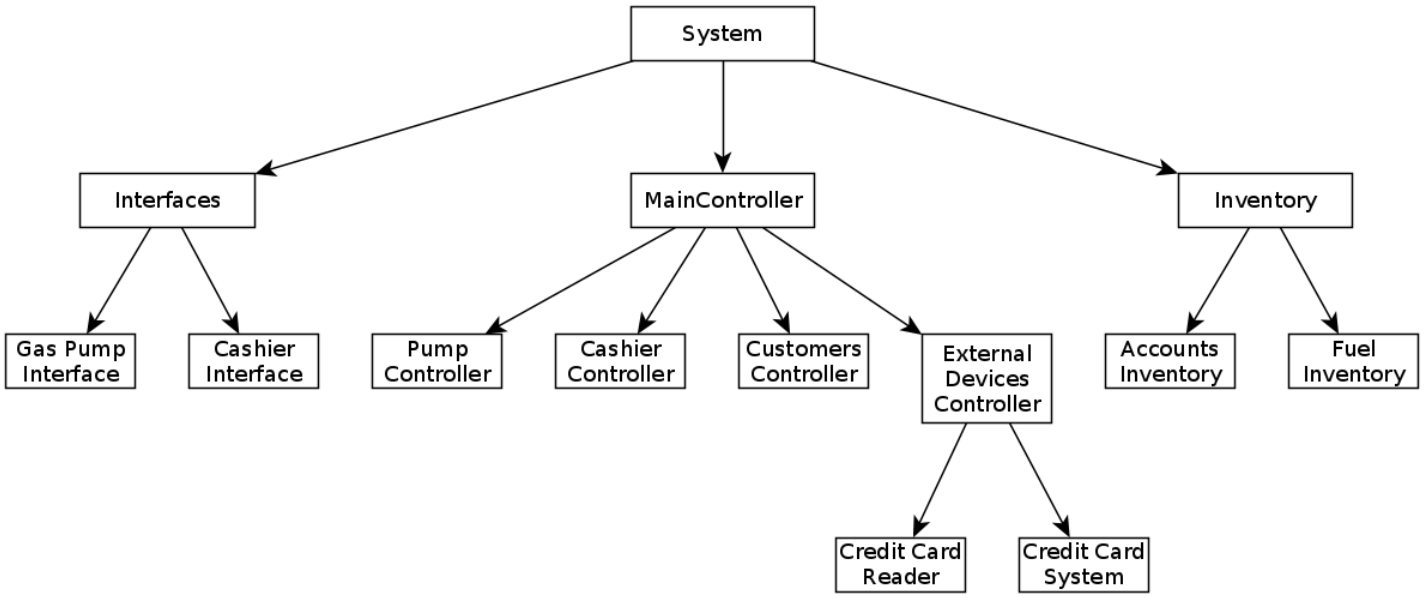
3 Design Patterns **6**
3.1 Template Method pattern . . . 6
3.2 Strategy pattern . . . 6
3.3 Decorator pattern . . . 7
3.4 Observer pattern . . . 7
3.5 Composite pattern . . . 8

Introduction

To provide a system for a gas pump, we continue the analyse of the design. We will see more in details the architectural design, the classes and uses diagrams and several design patters.

1 Architectural Design

1.1 Hierarchical decomposition



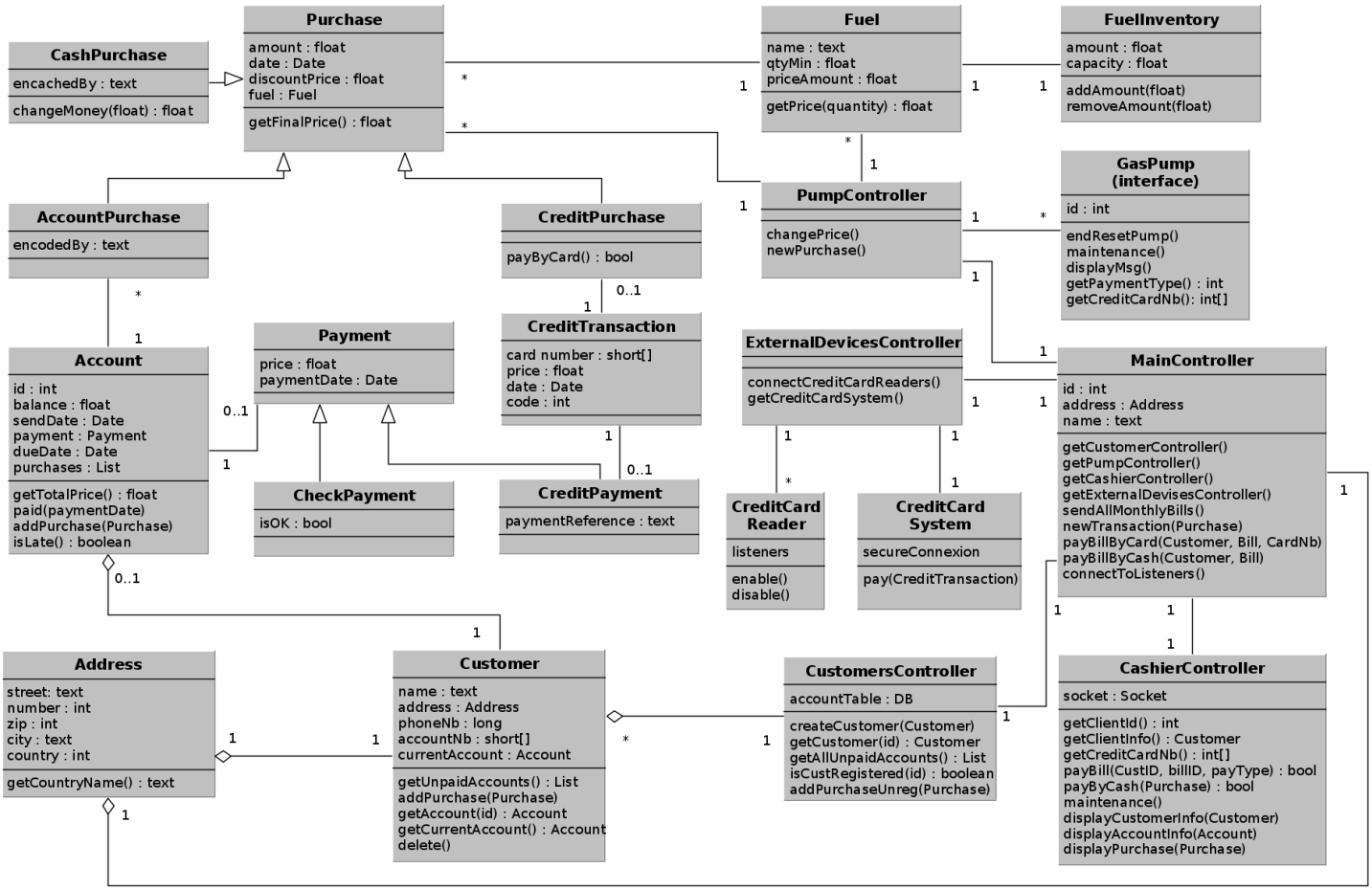
1.2 Roles and interactions

All componnents are managed by the system which is here at the top of the hierarchical decomposition schema. Then we have several parts:

- `Interfaces`: the interfaces globally display messages and wait for instructions. There are two graphical interfaces: the one of the Gas Pump and the one of the cashier (which is running on the same computer as the GSC).
- `MainController`: we decided to use controllers to manage different parts:
  - Pump Controller: this controller manages all things related to the pump: the gas pump system and interface and the inventory of the gas. For the payment, there is also a link with the external devises and the cashier controller.
  - Cashier Controller: it is responsible of the cashier interface and the account inventory. For the payment, it will also be linked with the external devises and the customer controller.
  - Customers Controller: this controller manages all things related to the accounts: the bills and clients inventory.
  - External Devises Controller: it is responsible of the external devises: the credit card system and reader. The external devises use specific communication protocols implemented in available software drivers and libraries. This controller will do the link between these devises and the system.
- `Inventory`: The inventories of the accounts and the stock (fuel/gas) are stored in specific databases.

2 Detailed Design

2.1 Class diagram



2.2 Short description of each class

2.2.1 `MainController`

The main controller manages all the system, it's the core of the application. From it, it's possible to get all other controllers.

This controller will initialise all other controllers and it will do the link between them.

2.2.2 `CashierController`

This controller will be in charge of the cashier interface: it will be able to react to signals sent from this interface and send messages to it to display informations requested by the cashier or the client through the gas pump interface.

2.2.3 `PumpController`

This class is responsible of the pumps, their interfaces and the fuel. Like the controller of the cashier, it will be able to react to signals sent from the gas pump interfaces (i.e. a new purchase is available, it will prepare the `Purchase` object and delegate it to the `CashierController` if the user want to add this purchase to its monthly bill).
From this controller, we will also be able to change price of the fuel.

2.2.4 GasPump

The main goal of this class is to encapsulate the communication with the pump and its interface by using specific communication protocols. Then it will display appropriated messages for the client and reset the counter of the pump when the transaction is over.

2.2.5 Fuel

This class represents a sort of fuel that can be bought by a client. It has a name (e.g. Diesel), a minimum quantity that can be taken (e.g. 2.5 gallons) and the price (3.596$ for 1 gallon).

2.2.6 FuelInventory

Linked to a Fuel object, this inventory is used to know the status of the stock with the current quantity. This amount has to be updated by the PumpController after each refueling by a client.

2.2.7 Purchase

A purchase is created after each refueling. It contains all informations needed for a purchase (fuel, quantity, price, date). A purchase can be paid by three different ways: by cash, by adding it on a monthly bill or by credit card payment.

2.2.8 CashPurchase

This class represents a payment by cash. This cash payment has been received by a cashier.

2.2.9 AccountPurchase

A purchase can be added to an monthly bill and then we need a specific AccountPurchase object. We also need to remember the name of the cashier who is responsible of this purchase.

2.2.10 CreditPurchase

When the purchase is directly paid with a credit card, we need a specific CreditPurchase object to easily manage a credit card transaction.

2.2.11 CustomersController

The controller of the customers is responsible of the customers and the inventory of the accounts. All purchases have to be saved even the one from unregistered users. This class is also used to create new customer and to get all unpaid accounts.

2.2.12 Customer

This class represents the registered customer with all info linked him/her. We can add purchase to his/her current monthly bill from this object.

2.2.13 Address

It's a basic object used by the MainController and the Customer to represent an physical address.

2.2.14 Account

This class represents a bill with all details linked to it (id, purchases, date, balance, etc.). A payment can be associated to this bill if it's already been paid.

2.2.15 `Payment`

This class represents a payment that can be made by cash or by credit card. In addition to the price, the payment object also contains a date.

2.2.16 `CheckPayment`

This class is just used to check that the payment is correct (the right price, the right data, etc.).

2.2.17 `CreditPayment`

This one represents a payment with a credit card. References about the payment need to be stored.

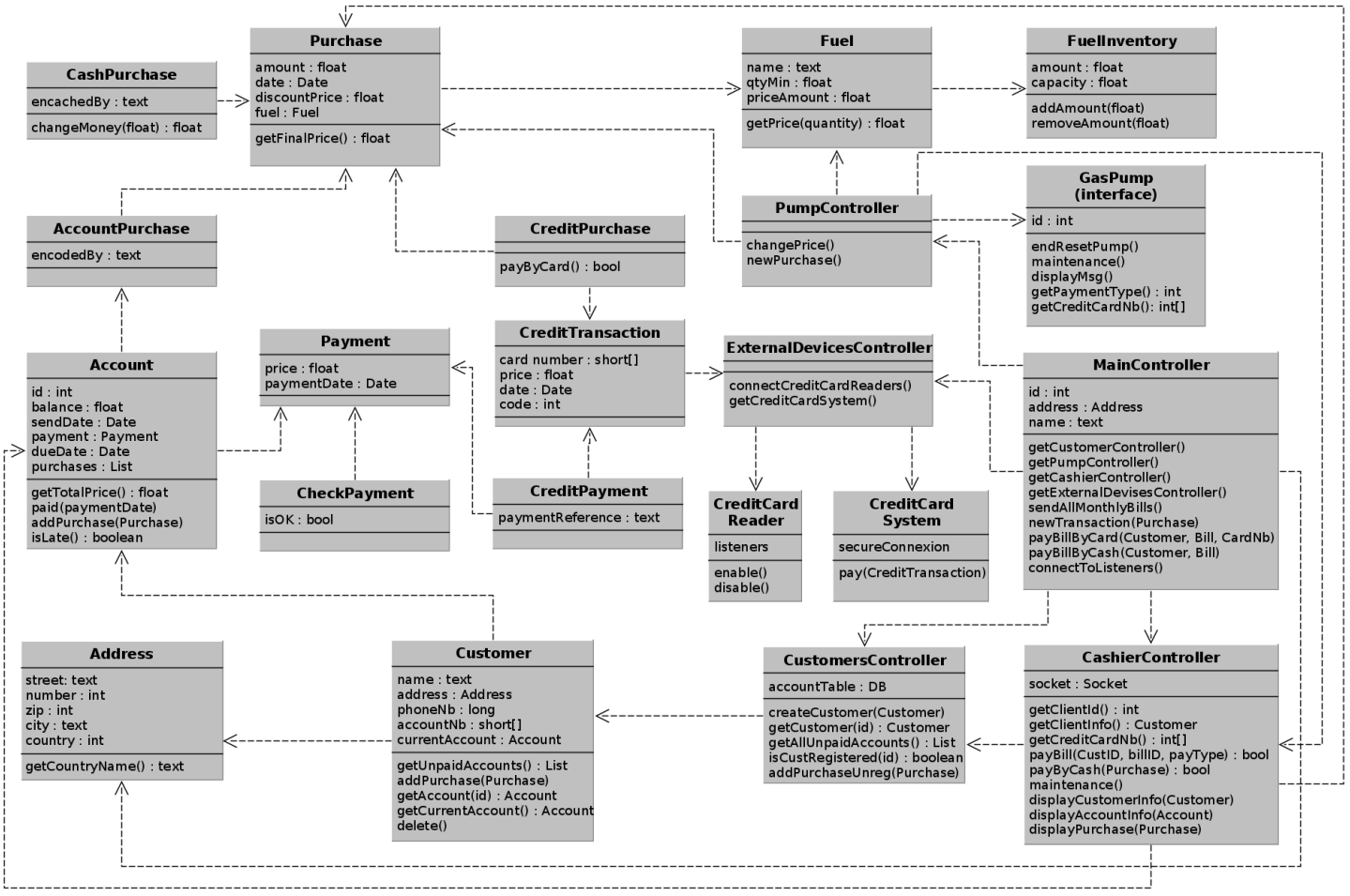
2.2.18 `CreditTransaction`

This class is used to do the link between the data and the credit card reader and system. It will temporary save all data needed for a credit card transaction.

2.2.19 `ExternalDevicesController`

This controller is used to manage external devices which use specific protocols. Currently, there is two external devices: the credit card reader (to read a credit card and accept the code) and the credit card system which is used to communicate with the bank system via a secure connexion.

2.3 USES diagram



With this USES diagram, we can see that we have no loop in our program. Note that some possible loops have been eliminated when controllers are communicated between each other. In reality, one controller (e.g. `PumpController`) accesses to another (e.g. `CashierController`) via the `MainController`: it uses this main controller just to get another controller, nothing more.

According to this diagram, here is the possible sets of classes for an incremental development:

- Address, Fuel and FuelInventory, CreditCardReader, CreditCardSystem, GasPump, Payment.
- Purchase, ExternalDevicesController, CheckPayment.
- CreditTransaction, AccountPurchase, CashPurchase.
- CreditPayment, CreditPurchase, Account.
- Customer and CustomersController.
- CashierController and PumpController.
- MainController.

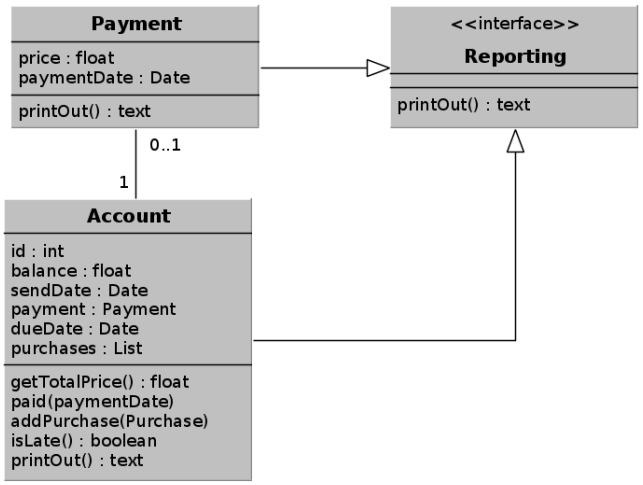
3 Design Patterns

3.1 Template Method pattern

Here is the method `getReceipt()` from the `Payment` class:

```
public String getReceipt ()
{
        String output = "Dinoco GSCS\n"
        output += "Date: " + getPaymentDate () + "\n";
        output += "Price: " + getPrice () + "\n";
        output += "Thank you!"
        return output;
}
```

3.2 Strategy pattern



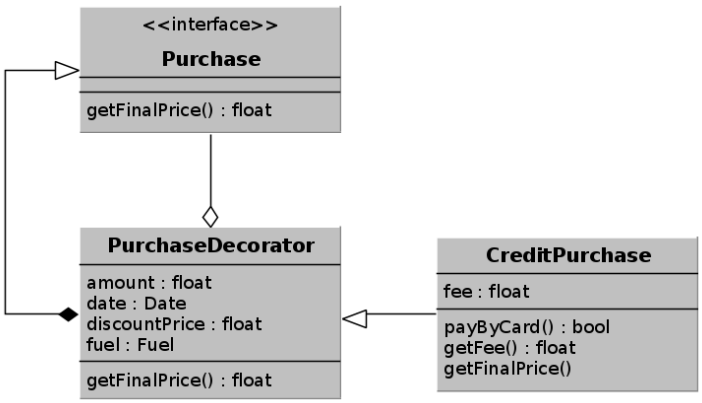
Here is the Java code added to the `Payment` class:

```
public String printOut ()
{
        String output = "Payd. Details:\n"
        output += "Date: " + getPaymentDate () + "\n";
        output += "Price: " + getPrice () + "\n";
        return output;
}
```

And for the `Account` class:

```
public String printOut ()
{
        String output = "Account #" + id + ":\n";
    if (isLate ())
        output += "! Warning ! ";
        output += "Due Date: " + getDueDate () + "\n";
        output += "Price: " + getTotalPrice () + "\n";
    if (payment != null)
            output += payment.printOut();
        return output;
}
```

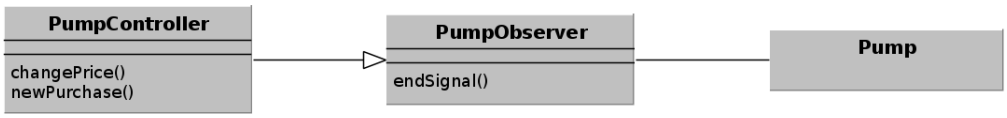
3.3 Decorator pattern



The `getFinalPrice()` method of the `CreditPurchase` method can be defined like that.

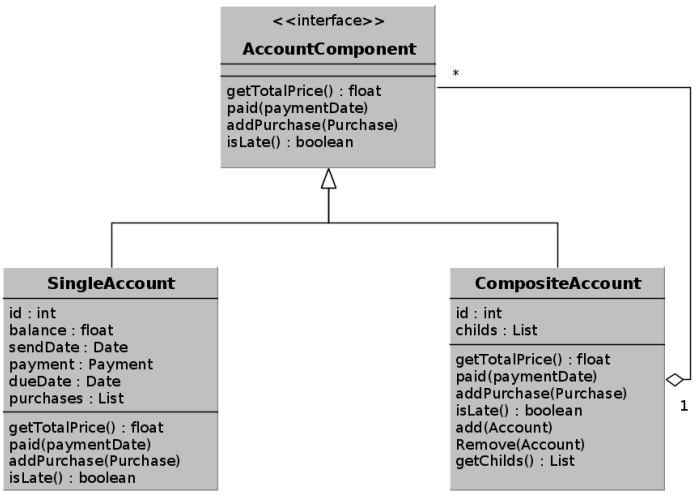
```
@override
public float getFinalPrice ()
{
        return super.getFinalPrice() + this.getFee();
}
```

3.4 Observer pattern



We decided to add two new classes to respect the Observer pattern: `GasPumObserver` which is observing the nozzle of the class `Pump` is used by the customer. This first class will notify the change to the `PumpController`.

3.5 Composite pattern



We decided to split the `Account` class into 3 parts.

- `AccountComponant`: it's an interface and represents the component.
- `SingleAccount`: it represents one single account and implements all methods of the interface
- `CompositeAccount`: it also implements all methods of the interface and it contains all methods to manipulate children.

Conclusion

With this new assignment, we saw in more details the analyse of a architectural design and different design patterns.
It was a good exercise to improve our vision of a real case.