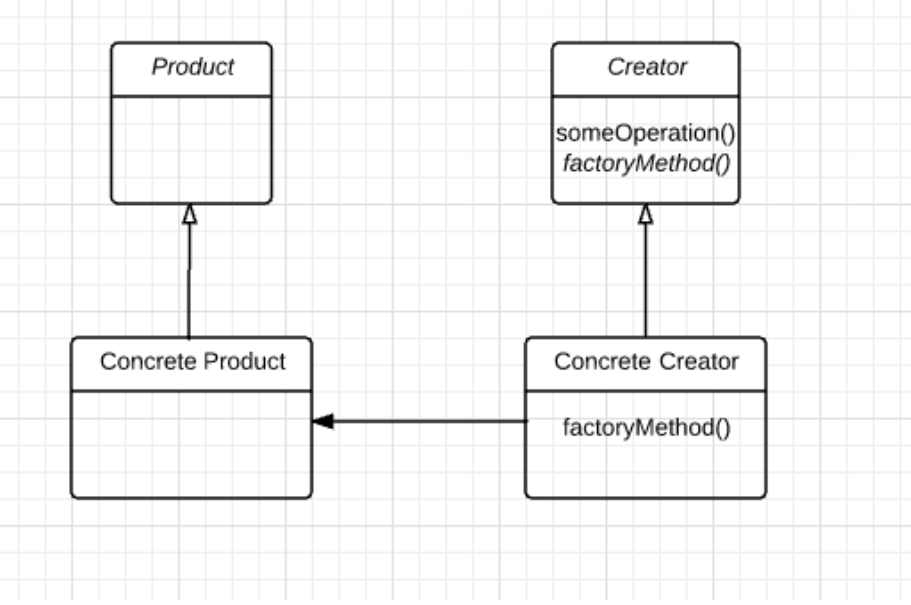
Design Patterns

COURSE NOTES

Version 0.1.0



Façade Pattern

As systems or parts of systems become larger, they also become more complex. This is not necessarily a bad thing – if the scope of a problem is large, it may require a complex solution. Client classes function better with a simpler interaction, however. The **façade design pattern** attempts to resolve this issue, by providing a single, simplified interface for client classes to interact with a subsystem. It is a structural design pattern.

A **façade** is a wrapper class that encapsulates a subsystem in order to hide the subsystem's complexity, and acts as a point of entry into a subsystem without adding more functionality in itself. The wrapper class allows a client class to interact with the subsystem through the façade. A façade might be compared metaphorically to a waiter or salesperson, who hide all the extra work to be done in order to purchase a good or service.

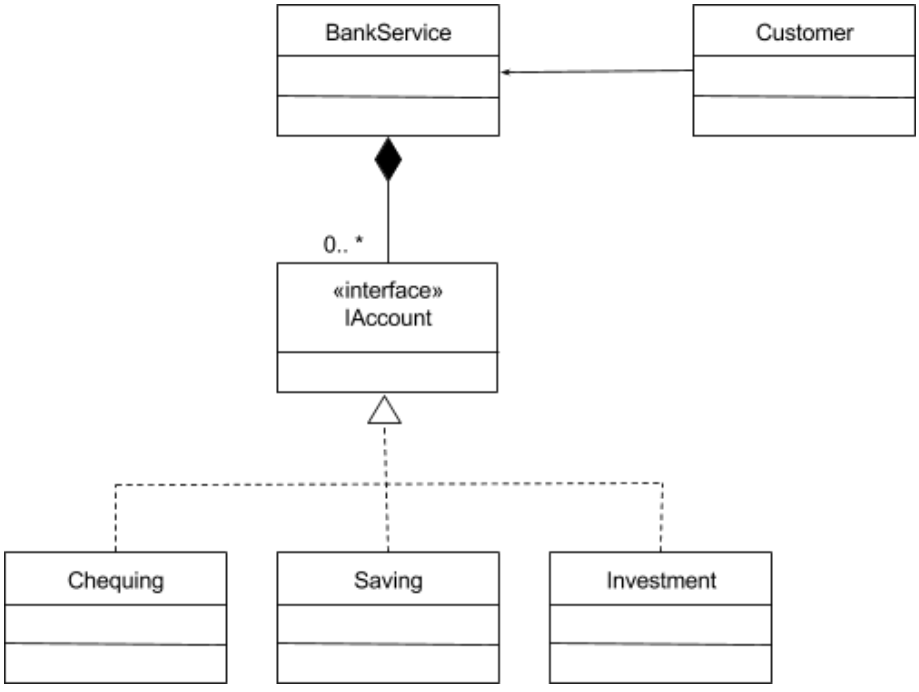
A façade design pattern should therefore be used if there is a need to

simplify the interaction with a subsystem for client classes, and if there is a need for a class to instantiate other classes within your system and to provide these instances to another class. Often façade design patterns combine interface implementation by one or more classes, which then gets wrapped by the façade class. This can be explained through a number of steps.

1. Design the interface
2. Implement the interface with one or more classes
3. Create the façade class and wrap the classes that implement the interface
4. Use the façade class to access the subsystem

Let us examine each of these steps with an example for a bank system.

In the UML diagram below, we can see that a BankService class acts as a façade for Chequing, Saving, and Investment classes. As all three accounts implement the iAccount interface, the BankService class wraps the account interface and classes, presenting a simpler "front" for the Customer client class.



Let us look at how to do this with code for each step outlined above.

Step 1: Design the Interface

First, create an interface that will be implemented by the different account classes, but will not be known to the Customer class.

```
public interface IAccount {
	public void deposit(BigDecimal amount);
	public void withdraw(BigDecimal amount);
	public void transfer(BigDecimal amount);
	public int getAccountNumber();
}
```

Step 2: Implement the Interface with one or more classes

Implement the interface with classes that will be wrapped with the façade class. Note that in this simple example, only one interface is being implemented and hidden, but in practice, a façade class can be used to wrap all the interfaces and classes for a subsystem.

> Instructor's Note: Remember that interfaces allow the creation of subtypes! This means that in this example, Chequing, Saving, and Investment are subtypes of IAccount, and are expected to behave like an account type.

```
public class Chequing implements IAccount { … }
public class Saving implements IAccount { … }
public class Investment implements IAccount { … }
```

Step 3: Create the façade class and wrap the classes that implement the interface

The BankService class is the façade. Its public methods are simple to use and show no hint of the underlying interface and implementing classes. The **information hiding** principle is used here to prevent client classes from "seeing" the account objects, and how these accounts behave – note that access modifies for each Account have been set to private.

```
public class BankService {
	private Hashtable<int, IAccount> bankAccounts;


	public BankService() {
		this.bankAccounts = new Hashtable<int,
IAccount>
	}

	public int createNewAccount(String type,
BigDecimal initAmount) {
```

```
            IAccount newAccount = null;
            switch (type) {
                  case "chequing":
                        newAccount = new
Chequing(initAmount);
                        break;
                  case "saving":
                        newAccount = new
Saving(initAmount);
                        break;
                  case "investment":
                        newAccount = new
                  Investment(initAmount);
                        break;
                  default:
                        System.out.println("Invalid
                  account type");
                        break;
            }
            if (newAccount != null) {
                  this.bankAccounts.put(newAccount.getAc
            countNumber(), newAccount);
                  return newAccount.getAccountNumber();
            }
            return -1;
      }

      public void transferMoney(int to, int from,
BigDecimal amount) {
            IAccount toAccount =
this.bankAccounts.get(to);
            IAccount fromAccount =
this.bankAccounts.get(from);
            fromAccount.transfer(toAccount, amount);
      }
}
```

Step 4: Use the façade class to access the subsystem

With the façade class in place, the client class can access accounts through the methods of the BankService class. The BankService class will tell the client what type of actions it will allow the client to call upon, and then will delegate that action to the appropriate Account object.

```
public class Customer {
      public static void main(String args[]) {
            BankService myBankService = new
BankService();
            int mySaving =
myBankService.createNewAccount("saving", new
BigDecimal(500.00));
            int myInvestment =
myBankService.createNewAccount("investment", new
```

```
BigDecimal(1000.00));
            myBankService.transferMoney(mySaving,
myInvestment, new BigDecimal(300.00));
      }
}
```

Façade design patterns draw on a number of different design principles. Subsystem classes are encapsulated into a façade class. Encapsulation is also demonstrated through information hiding subsystem classes from client classes. This also represents a separation of concerns.

In summary, the façade design pattern:

- Is a means to hide the complexity of a subsystem by encapsulating it behind a unifying wrapper called a façade class.
- Removes the need for client classes to manage a subsystem on their own, resulting in less coupling between the subsystem and the client classes.
- Handles instantiation and redirection of tasks to the appropriate class within the subsystem.
- Provides client classes with a simplified interface for the subsystem.
- Acts simply as a point of entry to a subsystem and does not add more functional the subsystem.

-
-
-