Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Gang of Four's Pattern Catalog

Creational	Structural	Behavioral
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Facade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Facade Pattern: Intent

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Facade Pattern: Applicability

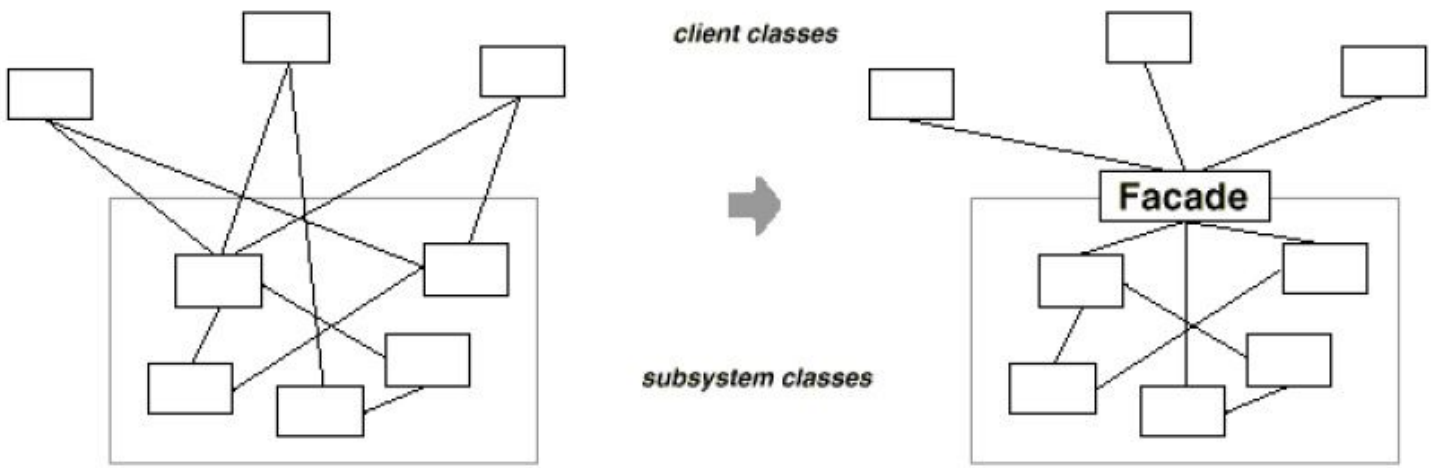
- Use the Facade pattern when
  - you want **to provide a simple interface to a complex subsystem**. Subsystems often get more complex as they evolve.
  - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby **promoting subsystem independence and portability**.
  - you want **to layer your subsystems**. Use a facade **to define an entry point to each subsystem level**.

Facade Pattern: Motivation

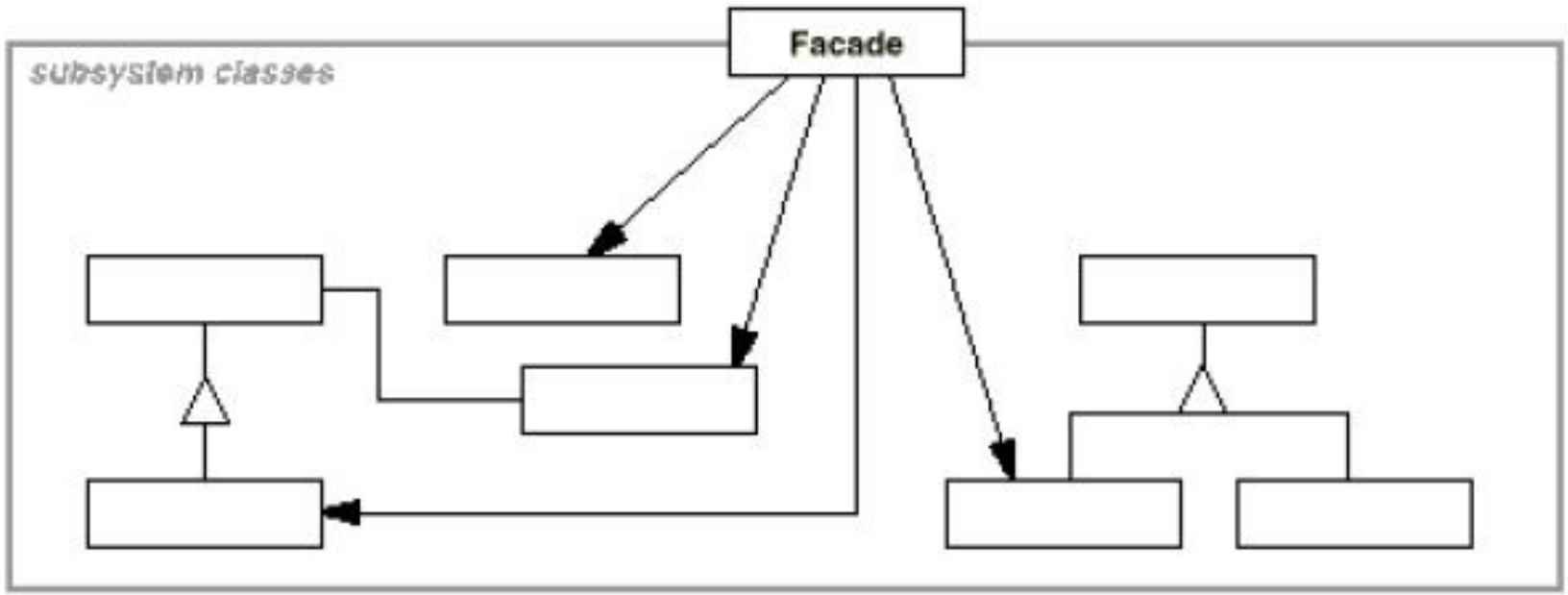
- Structuring a system into subsystems helps reduce complexity.

- A common **design goal** is to minimize the communication and dependencies between subsystems.

- One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.



Facade Pattern: Structure



Facade Pattern

- A facade 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 facade.

- A facade 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.

- Often facade design patterns combine interface implementation by one or more classes, which then gets wrapped by the facade class.

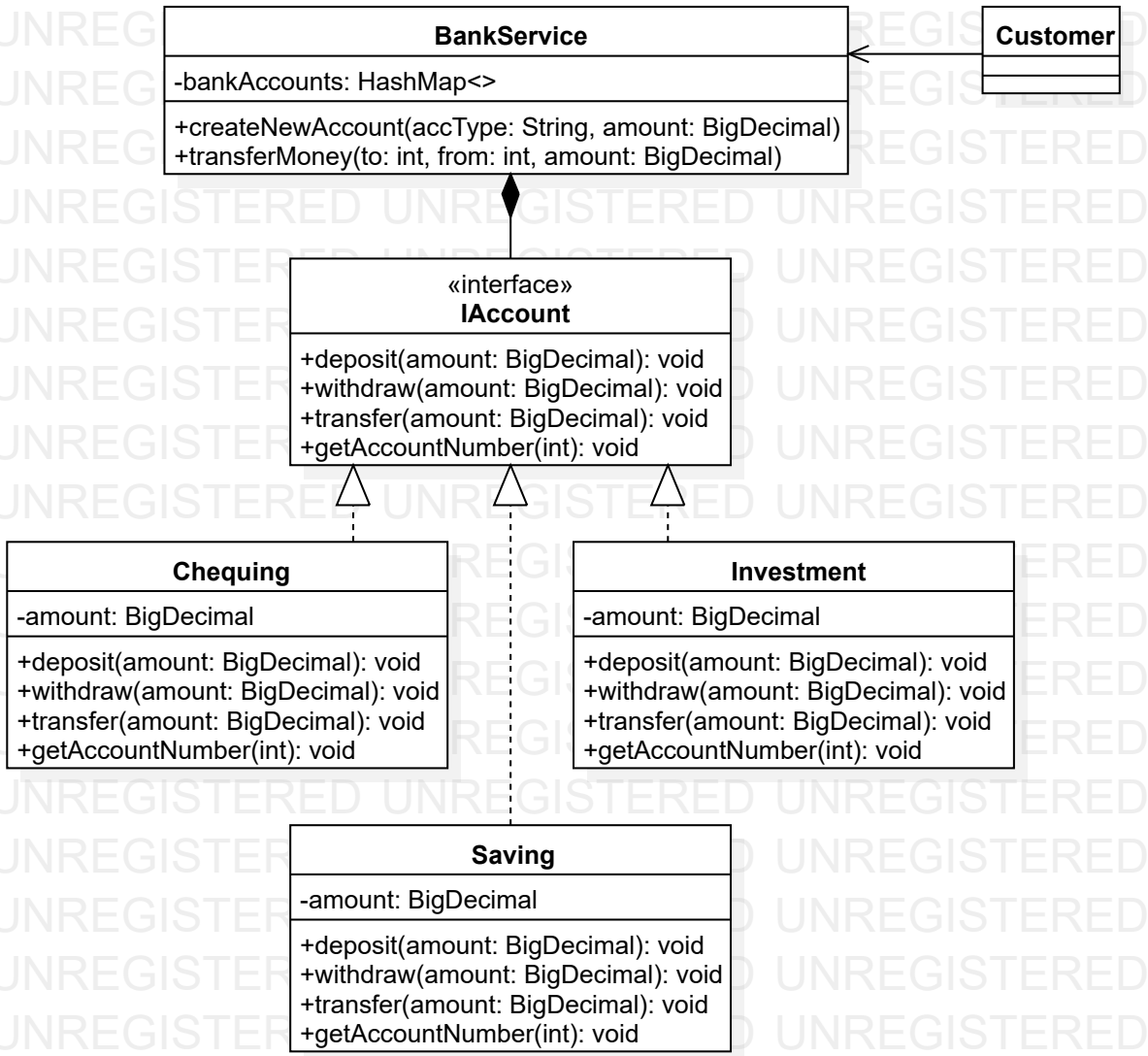
Facade Pattern (contd.)

- 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 facade class and wrap the classes that implement the interface
    4. Use the facade class to access the subsystem

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

Facade Pattern: Example



Facade Pattern: Example

- Step 1: Design the Interface

```
//IAccount
import java.math.BigDecimal;

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

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes

```
//Chequing
import java.math.BigDecimal;

public class Chequing implements IAccount
{
    private BigDecimal amount;
    public Chequing(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from
            CHEQUING account!");
    }
```

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Chequing (contd.)
    @Override
    public void withdraw(BigDecimal amount) {
        System.out.println("Hi, This is WITHDRAW from
            CHEQUING account!");
    }

    @Override
    public void transfer(BigDecimal amount) {
        System.out.println("Hi, This is TRANSFER from
            CHEQUING account!");
    }

    @Override
    public int getAccountNumber() {
        return 100001;
    }
}
```

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Saving
import java.math.BigDecimal;

public class Saving implements IAccount
{
    private BigDecimal amount;
    public Saving(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from SAVING
            account!");
    }
```

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Saving (contd.)
    @Override
    public void withdraw(BigDecimal amount) {
        System.out.println("Hi, This is WITHDRAW from SAVING
            account!");
    }

    @Override
    public void transfer(BigDecimal amount) {
        System.out.println("Hi, This is TRANSFER from SAVING
            account!");
    }

    @Override
    public int getAccountNumber() {
        return 200001;
    }
}
```

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Investment
import java.math.BigDecimal;

public class Investment implements IAccount
{
    private BigDecimal amount;
    public Investment(BigDecimal initAmount) {
        this.amount = initAmount;
    }

    @Override
    public void deposit(BigDecimal amount) {
        System.out.println("Hi, This is DEPOSIT from
            INVESTMENT!");
    }
```

Facade Pattern: Example (contd.)

- Step 2: Implement the Interface with one or more classes (contd.)

```
//Investment (contd.)
    @Override
    public void withdraw(BigDecimal amount) {
        System.out.println("Hi, This is WITHDRAW from
            INVESTMENT!");
    }

    @Override
    public void transfer(BigDecimal amount) {
        System.out.println("Hi, This is TRANSFER from
            INVESTMENT!");
    }

    @Override
    public int getAccountNumber() {
        return 300001;
    }
}
```

Facade Pattern: Example (contd.)

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

```
import java.math.BigDecimal;
import java.util.HashMap;
//BankService
public class BankService
{
    private HashMap<Integer, IAccount> bankAccounts;

    public BankService()
    {
        this.bankAccounts = new HashMap<>();
    }
```

Facade Pattern: Example (contd.)

- Step 3: Create the facade class and wrap the classes that implement the interface (contd.)

```
//BankService (contd)
    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;
        }
```

Facade Pattern: Example (contd.)

- Step 3: Create the facade class and wrap the classes that implement the interface (contd.)

```
//BankService (contd)
        if (newAccount != null)
        {
            this.bankAccounts.put(newAccount.getAccountNumber(),
                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);
    }
}
```

Facade Pattern: Example (contd.)

- Step 4: Use the facade class to access the subsystem

```
//Customer
import java.math.BigDecimal;

public class Customer
{
    public static void main(String[] args)
    {
        BankService myBankService = new BankService();
        int mySaving =
            myBankService.createNewAccount("saving", new
            BigDecimal(500.00));
        System.out.println("New saving account created with
            account number: "+mySaving);
        int myInvestment =
            myBankService.createNewAccount("investment",
            new BigDecimal(1000.00));
        System.out.println("New investment account created
            with account number: "+myInvestment);
```

Facade Pattern: Example (contd.)

- Step 4: Use the facade class to access the subsystem (contd.)

```
//Customer (contd)
        myBankService.transferMoney(mySaving, myInvestment,
            new BigDecimal(300.00));
        System.out.println("Money transfered from saving
            account "+mySaving+" to investment account: "+
            myInvestment);
    }
}
```

Facade Pattern: Summary

- Is a means to hide the complexity of a subsystem by encapsulating it behind a unifying wrapper called a facade 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.