Design Patterns em Java

Command

Facade

Command design pattern

- O padrão *Command* permite encapsular, através de classes de objetos, os comandos a serem executados sobre um determinado alvo – o recetor dos comandos
- Cada comando define as operações a serem executadas quando é executado através de um gestor de comandos
- Adicionalmente, caso seja possível, pode-se definir o modo de "desfazer" a operação realizada (operação de *undo*)
  - Caso as operações de *undo* sejam facultadas para os comandos, o gestor pode disponibilizar as funcionalidades de *undo* e *redo*

ICommand

- `ICommand` – Interface base para todos os comandos

```
public interface ICommand {
    boolean execute();
    boolean undo();
}
```

  - A ação de *undo* pode não se justificar em determinados contextos
    - O método `execute` deverá retornar `true` caso a operação possa ser alvo de uma ação de *undo*

CommandAdapter

- `CommandAdapter` (opcional) – classe que permite disponibilizar implementações comuns para os comandos
  - Por exemplo, manter uma referência para o recetor dos comandos

```
abstract class CommandAdapter implements ICommand {
    protected ReceiverClass receiver;

    protected CommandAdapter(ReceiverClass receiver) {
        this.receiver = receiver;
    }
}
```

ConcreteCommand

- Cada classe que deriva de `CommandAdapter` corresponderá, normalmente, a um comando concreto que poderá ser executado e, eventualmente, desfeito (*undo*)

```
public class ConcreteCommand extends CommandAdapter {
    private <variables needed to execute this command>

    public ConcreteCommand(ReceiverClass receiver, <additional parameters>) {
        super(receiver);
        // store parameters
    }

    @Override
    public boolean execute() {
        return receiver.concreteCommand(<parameters>);
    }

    @Override
    public boolean undo() {
        return receiver.undoConcreteCommand(<parameters>);
    }
}
```

Invoker / CommandManager

- O gestor de comandos será responsável por ...
  - executar o comando (chamar o método `execute`)
  - gerir o histórico de comandos para possibilitar operações de *undo* (se disponíveis)
  - gerir uma lista de comandos para realizar operações de *redo*

- Tendo em consideração o funcionamento típico das operações de *undo* e *redo*, a sua gestão poderá ser auxiliada por objetos `Stack` ou `Deque`
  - Disponibilizam funções *push* e *pop*

Invoker / CommandManager

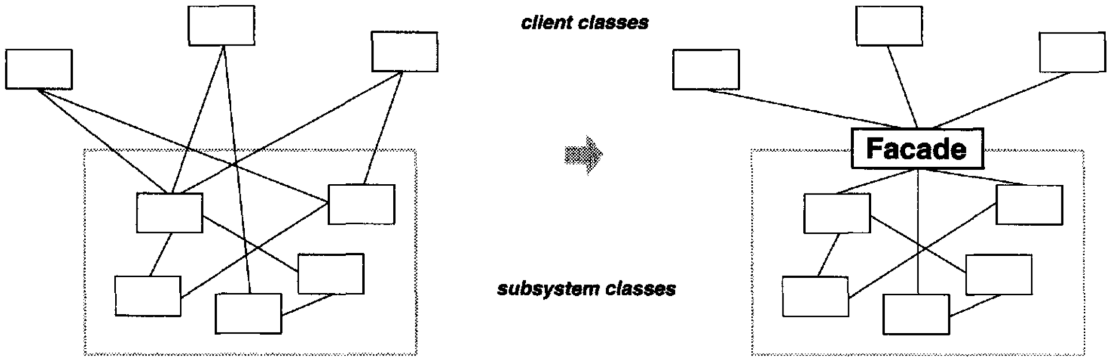
```
public class CommandManager {
    private Deque<ICommand> history;
    private Deque<ICommand> redoCmds;
    //private Stack<ICommand> history;
    //private Stack<ICommand> redoCmds;

    public CommandManager() {
        history = new ArrayDeque<>();
        redoCmds = new ArrayDeque<>();
        //history = new Stack<>();
        //redoCmds = new Stack<>();
    }

    public boolean invokeCommand(ICommand cmd) {
        redoCmds.clear();
        if (cmd.execute()) {
            history.push(cmd);
            return true;
        }
        history.clear();
        return false;
    }

// ... => ...
```

```
// ... => ...

    public boolean undo() {
        if (history.isEmpty())
            return false;
        ICommand cmd = history.pop();
        cmd.undo();
        redoCmds.push(cmd);
        return true;
    }

    public boolean redo() {
        if (redoCmds.isEmpty())
            return false;
        ICommand cmd = redoCmds.pop();
        cmd.execute();
        history.push(cmd);
        return true;
    }

    public boolean hasUndo() {
        return history.size()>0;
    }

    public boolean hasRedo() {
        return redoCmds.size()>0;
    }
}
```

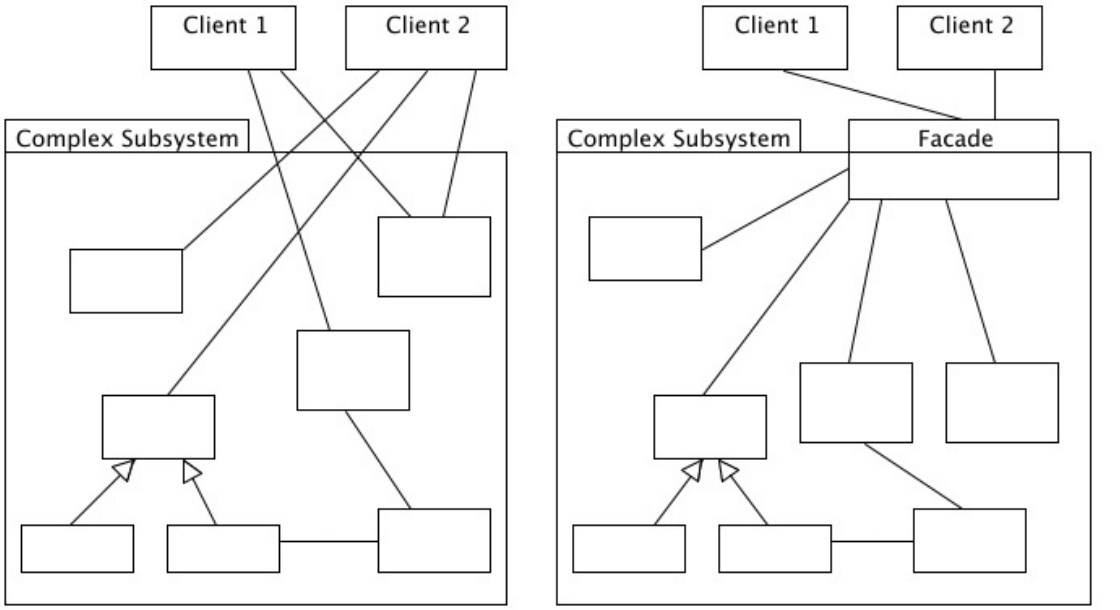
Facade

- A utilização do padrão *Command*, tal como de outros padrões ou implementações mais complexas, poderão e deverão ser escondidas das classes que vão usufruir das suas funcionalidades, por exemplo, das classes de interface com o utilizador
- Os pormenores de implementação poderão ser ofuscados colocando uma classe intermediária que servirá de ponto de acesso às funcionalidades disponibilizadas por esses sistemas
- A esta interface simplificada de acesso dá-se o nome de *Facade*

Facade



Erich Gamma et al., 1995, "Design Patterns", Addison-Wesley



http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/facade.html

Facade

- Uma classe *Facade* deverá fornecer métodos simples que permitam redirecionar as execuções pretendidas para os objetos internos dos *packages* (dos quais se pretende limitar a visibilidade)
  - Em algumas aplicações as classes *Facade* podem ser implementadas seguindo o padrão *Singleton*

Exemplo de *Facade* para *Command*

```
public class ReceiveManager { // Facade
    ReceiverClass rc;
    CommandManager cm;

    public ReceiveManager() {
        rc = new ReceiverClass();
        cm = new CommandManager();
    }

    public boolean concreteCommand1(<parameters>) {
        return cm.invokeCommand(new ConcreteCommand1(rc,<parameters>));
    }

    public boolean concreteCommand2(<parameters>) {
        return cm.invokeCommand(new ConcreteCommand2(rc,<parameters>));
    }

    // ...

    public boolean hasUndo() { return cm.hasUndo(); }

    public boolean undo()    { return cm.undo();    }

    public boolean hasRedo() { return cm.hasRedo(); }

    public boolean redo()    { return cm.redo();    }

    // ...
}
```

Exercício

- Desenvolva uma aplicação que permita gerir uma lista de compras, aplicando os padrões *Command* e *Facade*
  - Ações possíveis
    - Adicionar produto
    - Remover produto
    - *Undo*
    - *Redo*

Classe ShoppingList e Product

```
public class ShoppingList {
    private ArrayList<Product> list;

    public ShoppingList() {
        list = new ArrayList<>();
    }

    public boolean addProduct(String name, double qt) {
        if (name!=null && !name.isBlank() && qt>0) {
            list.add(new Product(name, qt));
            return true;
        }
        return false;
    }

    public boolean removeProduct(String name, double qt) {
        return list.remove(new Product(name,qt));
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder("Shopping List:\n");
        for(Product p : list)
            sb.append("\t- ").append(p).append("\n");
        return sb.toString();
    }
}
```

```
record Product(String name, double qt) {
    @Override
    public String toString() {
        return String.format(
                "%-20s %8.2f",
                name,qt
        );
    }
}
```

ShoppingListUI

```
public class ShoppingListUI {
    ShoppingListManager sm;

    public ShoppingListUI(ShoppingListManager sm) { this.sm = sm; }

    public void start() {
        boolean finish = false;
        do {
            System.out.println("\n\n"+sm+"\n");
            int op = PAInput.chooseOption("Shopping List",
                    "Add product","Remove product","Undo","Redo","Quit");
            switch (op) {
                case 1 -> sm.addProduct(
                            PAInput.readString("Product name: ",false),
                            PAInput.readNumber("Quantity: ") );
                case 2 -> sm.removeProduct(
                            PAInput.readString("Product name: ",false),
                            PAInput.readNumber("Quantity: ") );
                case 3 -> sm.undo();
                case 4 -> sm.redo();
                case 5 -> finish = true;
            }
        } while (!finish);
    }
}
```