# AA 19/20 – Relatório Design Patterns

João Marques (A81826), José Pereira (A82880), Ricardo Petronilho (A81744)

10 de Março de 2020

## 1 INTRODUÇÃO

O presente relatório relata a investigação sobre **Design Patterns** no âmbito da Unidade Curricular Arquitecturas Aplicacionais do 4º ano do Mestrado integrado em Engenharia Informática da Universidade do Minho.

O grupo de trabalho decidiu apresentar dois Design Patterns, sendo estes: o **Memento** e o **Observer**.

## 2 DESIGN PATTERN - MEMENTO

O **Memento** é um design pattern que permite ir guardando várias versões do estado ao longo do tempo, permitindo assim voltar a um estado anterior.

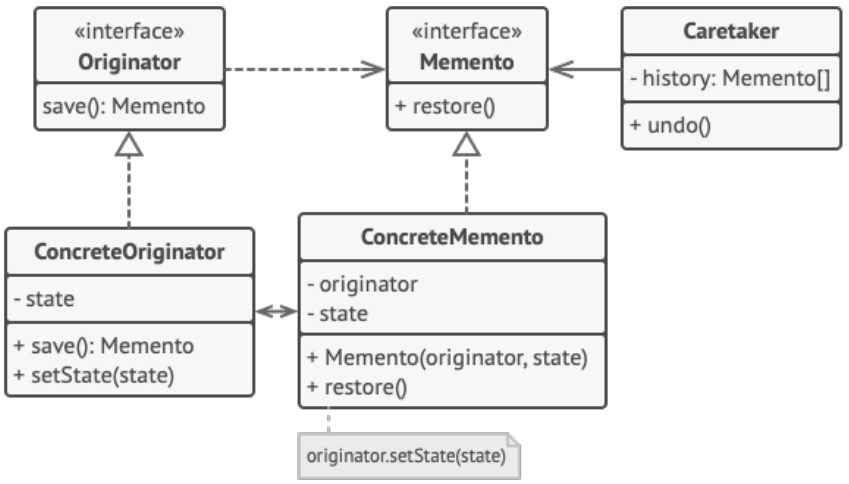


Figure 1. Arquitetura genérica do design pattern Memento.

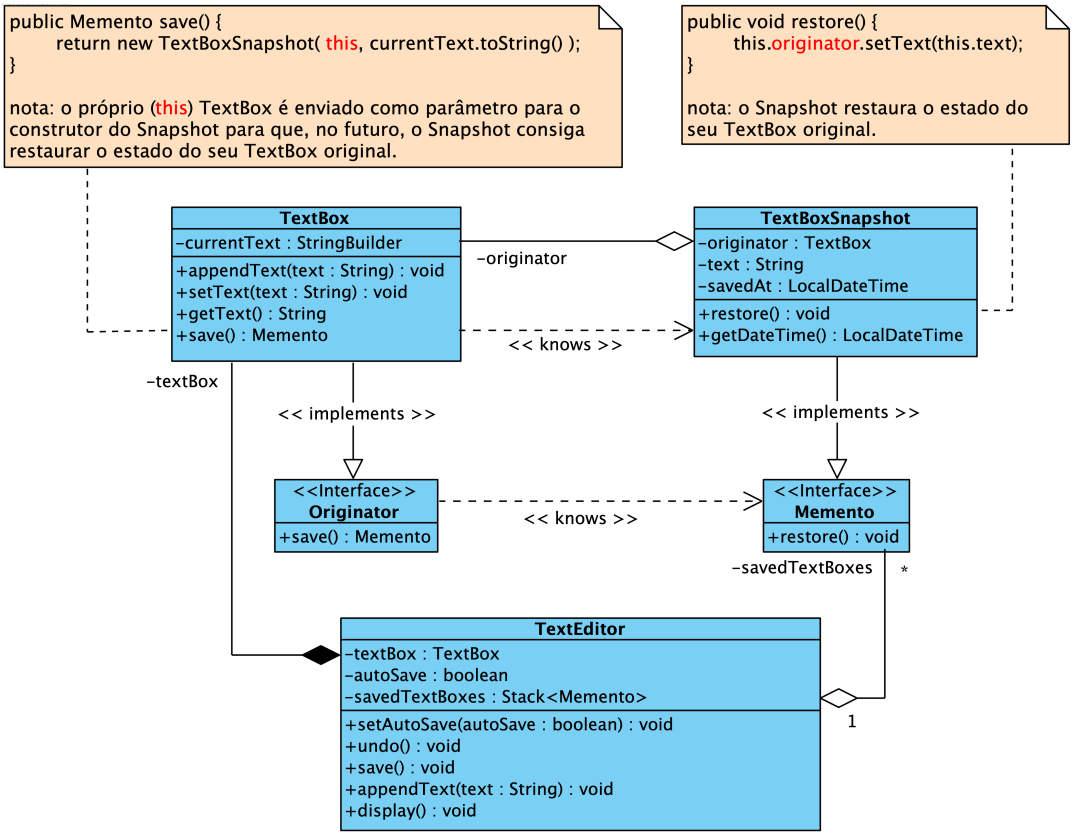


Figure 2. Arquitectura de um editor de texto usando o design pattern Memento.

Na figura acima está representada a arquitectura deste *design pattern* com um pequeno exemplo implementado.

Quando se quer gravar os novos estados gerados num **TextBox** existem duas formas de o fazer:

- invocando o método **save()** no **TextEditor** que cria um novo snapshot do estado atual e guarda-o na stack.

- outra forma opcional, utilizada neste exemplo em específico, é colocar a variável *autoSave* a *true*, assim sempre que é feita uma alteração ao estado é criado automaticamente um snapshot e adicionado à stack através da operação *push()*.

No TextEditor sempre que se quiser que os seus componentes voltem a um estado anterior basta invocar o método **undo()**, este método vai interagir com a variável *savedTextBoxes* que é uma *Stack* de **Memento** (instância de **TextBoxSnapshot**), ou seja, sempre que se pretende fazer o *undo()* será realizado um *pop()*, garantindo que se vai buscar o estado exactamente anterior ao momento da invocação. Depois de se obter o estado exactamente anterior ao estado actual é realizado o *restore()* à variável do tipo **Memento** que irá reverter o estado actual do TextBox associado à mesma.

Existem duas alternativas arquitecturais possíveis para reverter o estado do TextBox.

- 1ª alternativa: o próprio TextBox tem o método - **restore(TextBoxSnapshot s)** - que recebendo **qualquer** snapshot reverte o seu estado para o indicado no snapshot. Note-se que esta alternativa pode ter um **problema de integridade** uma vez que torna possível um TextBox reverter para um estado de **outro** TextBox, o que à partida apenas devia ser possível reverter para um estado que anteriormente foi do **próprio**.

- 2ª alternativa: no momento que o TextBox cria uma **TextBoxSnapshot** é enviado como parâmetro para este último o TextBox que o criou - **originator** - desta forma o método - **restore()** - encontra-se no TextBoxSnapshot, tornando-se impossível que um TextBox reverta o seu estado para outro que não seja anteriormente do próprio.

A arquitetura apresentada neste documento segue a 2ª alternativa. Note-se que apesar da 1ª alternativa manifestar um problema de integridade, pode ser usada propositadamente, o contexto do problema que se pretende resolver é que motiva os dois tipos de implementação.

Neste momento poderá estar a questionar-se sobre o motivo da criação de uma nova classe TextBoxSnapshot para guardar uma cópia do estado de um TextBox. A forma mais simples de o fazer seria, por exemplo, criar um método - **TextBox clone()** - que devolve um novo TextBox exatamente igual em valor. No entanto, dessa forma, o novo TextBox clonado não só guarda uma cópia do valor como também o **comportamento** e isso pode **não ser seguro**. Desta forma ao criar uma nova classe TextBoxSnapshot apenas se copia o estado e respetivamente os métodos get() para aceder ao estado, no entanto o **comportamento está propositadamente omisso**.

No anexo A pode-se visualizar a implementação do diagrama de classes apresentado em cima.

## 3 DESIGN PATTERN - OBSERVER

O design pattern **Observer** implementa "uma para muitas" dependências, isto é, um objecto ("*the subject*") "observado" por vários objectos observadores/dependentes (*"observers"*). Desta forma, todos os *observers* são automaticamente notificados, quando o *subject* modifica o seu estado.

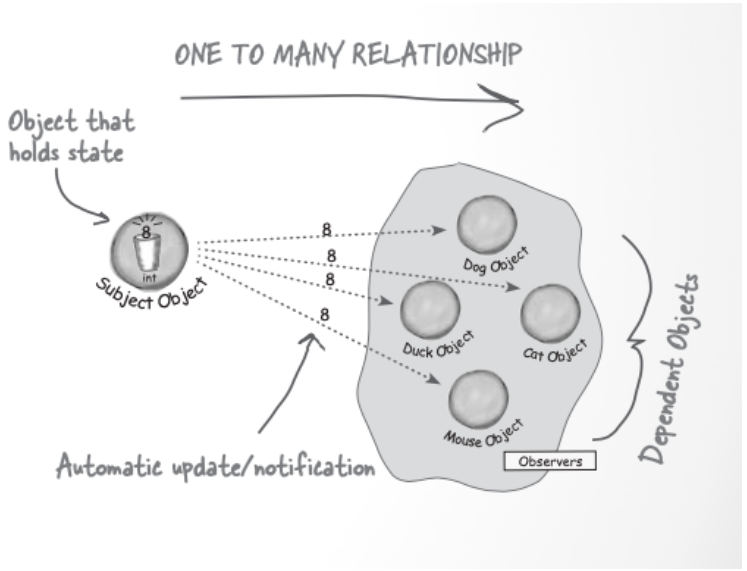


Figure 3. Ilustração do design pattern Observer.

O *subject* contém o *data model* (*state*) e/ou lógica de negócio, por outro lado, delega-se as funcionalidades da *view* para os *observers*. Desta forma, percebe-se que este design pattern pode ser usado em **MVC** (Model-View-Controller), no entanto, pode ser aplicado a outras situações.

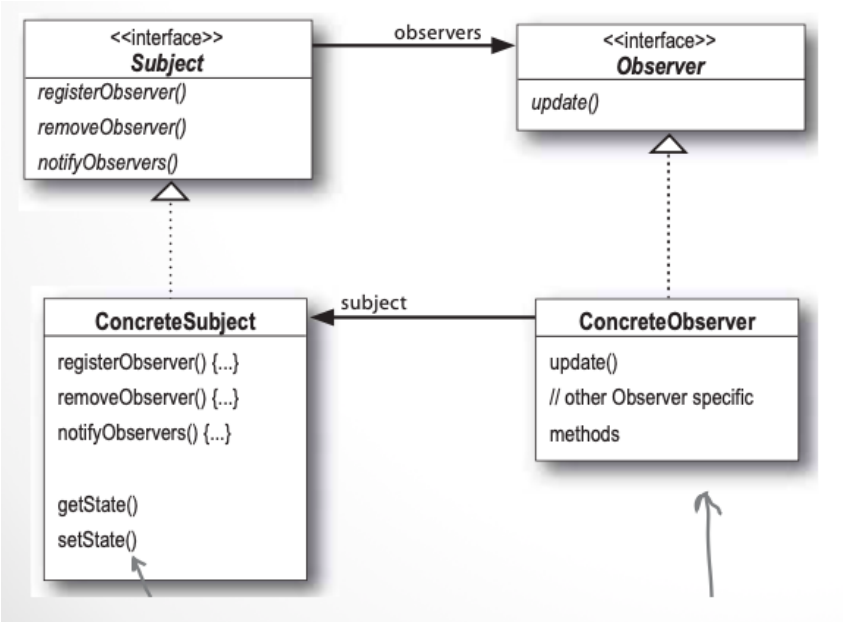


Figure 4. Arquitetura do design pattern Observer.

Os próprios *observers* registam-se como observadores do *subject*, quando têm interesse em serem notificados após uma alteração do estado deste, ou removem-se para deixar de receber essas notificações.

Com a finalidade de reutilização deste *design pattern*, para diferentes *subjects*, o mesmo sugere a criação de duas interfaces denominadas **Subject** e **Observer**, que delegam os métodos necessários para a concretização deste design pattern. Assim, para utilização deste design pattern, as classes observadas e observadoras devem implementar estas interfaces respetivamente.

Desta forma, e como foi dito anteriormente os *observers* registam-se/removem-se do subject, logo este comportamento irá ser definido no **ConcreteSubject**, classe que implementa a interface **Subject**, mas para ser usado/chamado na classe **ConcreteObserver**, classe essa, que implementa a interface **Observer**. Isto significa que, a **ConcreteObserver** vai ser composta por uma variável de instância do tipo **Subject**/**ConcreteSubject** (mais adiante perceber-se-á a razão de existir a opção de ser dos dois tipos), denominada na figura 4 por **subject**, para registar o *observer* no *subject* (registar o *observer*, consiste adicioná-lo a uma estrutura de dados definida na **ConcreteSubject**, explicada mais adiante). Ainda na classe **ConcreteObserver** define-se o método **update**, que será chamado quando o *subject* necessitar de notificar o *observer* e através deste enviar dados importantes para esta notificação.

Por outro lado, a classe **ConcreteSubject** vai ser constituída por uma estrutura de dados (p.e. *Collection*), que aglomere os observadores desse *subject*. Nesta classe, para além dos métodos de registo e remoção de observadores referidos anteriormente, existe ainda o método **notifyObservers**, que tal como o nome sugere, notificará os observadores, quando o estado do *subject* (**ConcreteSubject**) se alterar. O **notifyObservers** percorre a estrutura de dados que aglomera os *observers*, invocando o método **update**, definido nos mesmos, enviando os dados necessários.

Neste design pattern, verifica-se ainda uma das grandes vantagens do mesmo, que consiste na redução de dependências entre as classes concretas, que muitas vezes impedem a evolução do código. Desta forma, a classe **ConcreteSubject**, não depende da classe **ConcreteObserver** e vice versa, pois apenas conhecem/dependem das interfaces **Subject** e **Observer**.

Na verdade, poder-se-á pensar que o que foi afirmado antes, não é verdade, visto que na figura 4, existe uma dependência clara do **ConcreteObserver**, em relação ao **ConcreteSubject**. Realmente, esta dependência poderá acontecer, com o objectivo de facilitar o *observer* a aceder ao estado do *subject*. No entanto, como já se referiu anteriormente, podem ser enviados valores através do método **update**, e assim, não ser necessário criar esta dependência. Por outro lado, pode-se pensar que ao enviar dados pelo método **update**, este terá que ser específico a cada problema, e não se poderá reutilizar a interface **Observer**. No entanto, para resolver facilmente esta situação, coloca-se um argumento nos métodos **notifyObservers** e **update** do tipo **Object**, ou seja, pode-se enviar objectos de qualquer tipo, pois por exemplo na linaguagem *Java*, todo o objecto é do tipo **Object**.

Ainda, torna-se importante realçar, o facto de poder-se ter diferentes objectos observadores para o mesmo objecto observado. Quando diz-se diferentes, quer-se dizer, classes diferentes, no entanto, implementam a mesma interface, tornando-os do mesmo tipo. Assim, com esta vantagem torna-se possível por exemplo, de forma estruturada e organizada, ter um sensor de temperatura, classe observada, e diferentes formas de apresentação, classes observadoras (graus Celsius e Fahrenheit). Em relação ao exemplo prático apresentado em anexo B, apenas foi desenvolvida a classe **ConcreteObserver**, mas poderiam ter sido elaboradas diferentes classes que implementassem o **Observer** e apresentassem a informação de forma diferente.

A linguagem *Java* tem diversos design patterns implementados, e o **Observer** não é excepção, tendo este diversas implementações. Uma das primeiras, senão a primeira versão de implementação deste design pattern na ferramenta, não foi bem conseguida, sendo isso reconhecido pela própria

equipa do *Java*, tornando a classe em questão *deprecated*. A classe denomina-se por **Observable**, que "corresponde" ao **Subject** no caso apresentado acima. Desta forma, recomenda-se a não utilização do **Observable**, e fazer a interface **Subject** ou procurar por melhores alternativas, tais como **Listeners**.

Por outro lado, o *Java* definiu a interface **Observer**, não contendo qualquer problema, recomendando-se a sua utilização (no exemplo prático, definiu-se o **Observer**, mas poderia ter sido utilizado o do *Java*).

Em anexo B, apresenta-se um exemplo prático da aplicação deste design pattern. Utilizou-se, para facilitar a interpretação, os mesmos nomes para as classes e variáveis de instância da figura 4.

# 4 ANEXOS

## A MEMENTO

```java
public interface Originator {
    Memento save();
}

// ------------------------------------------------------------

public class TextBox implements Originator {

    private StringBuilder currentText;

    public TextBox() {
        currentText = new StringBuilder();
    }

    public void appendText(String text) {
        currentText.append(text);
    }

    public void setText(String text) {
        this.currentText.setLength(0); // more efficient than creating a new instance
        this.currentText.append(text);
    }

    public String getText() {
        return currentText.toString();
    }

    @Override
    public Memento save() {
        return new TextBoxSnapshot(this, currentText.toString());
    }
}

// ------------------------------------------------------------

public interface Memento {
    void restore();
}

// ------------------------------------------------------------

public class TextBoxSnapshot implements Memento {

    private TextBox originator;
```

```
    // cloned state variables from originator (TextBox)
    private String text;

    // custom variables for advanced features
    private LocalDateTime savedAt;

    public TextBoxSnapshot(TextBox originator, String text) {
        this.originator = originator;
        this.text = text;
        savedAt = LocalDateTime.now();
    }

    public LocalDateTime getDateTime() {
        return savedAt;
    }

    @Override
    public void restore() {
        originator.setText(text);
    }
}

// ------------------------------------------------------------

public class TextEditor {

    private Stack<Memento> savedTextBoxes; // stack holding snapshots
    private TextBox textBox; // current text box
    private boolean autoSave; // if true automatically
    // saves text box before making changes

    public TextEditor() {
        savedTextBoxes = new Stack<>();
        textBox = new TextBox();
        autoSave = false;
    }

    public void appendText(String text) {
        if (autoSave) save(); // creates a snapshot before making changes
        textBox.appendText(text);
    }

    public void setAutoSave(boolean autoSave) {
        this.autoSave = autoSave;
    }
    public void display() {
        System.out.println(">> " + textBox.getText());
    }
```

```
    public void save() {
        Memento snapshot = textBox.save(); // creates snapshot
        savedTextBoxes.push(snapshot); // saves it
    }

    public void undo() {
        System.out.println("undo clicked...");
        Memento snapshot = savedTextBoxes.pop();
        snapshot.restore();
    }
}

// ------------------------------------------------------------

public class Main {

    public static void main(String[] args) {
        TextEditor textEditor = new TextEditor();
        textEditor.setAutoSave(true);

        textEditor.appendText("Hello");
        textEditor.display();

        textEditor.appendText(" World!");
        textEditor.display();

        textEditor.undo();
        textEditor.display();
    }
}

// ------------------------------------------------------------

Output:

>> Hello
>> Hello World!
undo clicked...
>> Hello
```

## B OBSERVER

```java
public interface Subject {
    void registerObserver(Observer concreteObserver);
    void removeObserver(Observer observer);
    void notifyObservers(Object info);
}

// ------------------------------------------------------------
public interface Observer {
    void update(Object info);
}

// ------------------------------------------------------------

public class ConcreteSubject implements Subject {
    private int state;
    private Collection<Observer> observers;

    public ConcreteSubject(){
        this.state = 0;
        this.observers = new ArrayList<>();
    }

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(Object info) {
        System.out.println("notify observers");
        Object[] arr = new Object[2];   //  array with info to send
        arr[0] = this.state;            //  state
        arr[1] = info;                  //  time
        for(Observer ob : observers)ob.update(arr);
    }

    // other methods

    public int getState(){
        return this.state;
    }
```

```
    public void setState(int state){
        this.state = state;
    }
}

// ------------------------------------------------------------

public class ConcreteObserver implements Observer{
    private Subject subject;
    private int state;

    public ConcreteObserver(Subject subject){
        this.subject = subject;
        this.subject.registerObserver(this);
        //  observer registers on the subject
    }

    @Override
    public void update(Object info) {
        if(info instanceof Object[]){
            LocalDateTime time = null;
            Object[] arr = (Object[]) info;
            this.state = (Integer) arr[0];
            time = (LocalDateTime) arr[1];
             System.out.println("observer: state = " +
                                this.state +
                                " | time = "
                                + time);
        }else{
            System.out.println("Something is wrong!");
        }
    }

    // other methods
}

// ------------------------------------------------------------

public static void main(String[] args){
        int numberOfObservers = 3;

        //  subject
        ConcreteSubject subject = new ConcreteSubject();

        //  observer register
        ConcreteObserver obs = new ConcreteObserver(subject);
```

```
        //  create multiple observers
        for(int i = 0; i < numberOfObservers; i++) new ConcreteObserver(subject);

        System.out.println("initial state: " + subject.getState());

        //  the state is changed, then observers are notified
        subject.setState(50);

        System.out.println("change state value to: " + subject.getState());

        //  here, we are sending additional information to observers
        subject.notifyObservers(LocalDateTime.now());

        subject.setState(5741);
        System.out.println("change state value to: " + subject.getState());
        subject.notifyObservers(LocalDateTime.now());

        //  the status is changed and do not notify observers immediately and
        //  check that the status of observers is not updated
        subject.setState(-457);
        System.out.println("change state value to: " + subject.getState());
        System.out.println("state of observer before notifyObservers: " +
                           obs.getState());
        subject.notifyObservers(LocalDateTime.now());

        //  remove observer obs
        subject.removeObserver(obs);
    }

// ------------------------------------------------------------

Output:

initial state: 0
change state value to: 50
notify observers
observer: state = 50 | time = 2020-03-08T18:38:12.310
observer: state = 50 | time = 2020-03-08T18:38:12.310
observer: state = 50 | time = 2020-03-08T18:38:12.310
observer: state = 50 | time = 2020-03-08T18:38:12.310
change state value to: 5741
notify observers
observer: state = 5741 | time = 2020-03-08T18:38:12.328
observer: state = 5741 | time = 2020-03-08T18:38:12.328
observer: state = 5741 | time = 2020-03-08T18:38:12.328
observer: state = 5741 | time = 2020-03-08T18:38:12.328
change state value to: -457
state of observer before notifyObservers: 5741
```

```
notify observers
observer: state = -457 | time = 2020-03-08T18:38:12.329
observer: state = -457 | time = 2020-03-08T18:38:12.329
observer: state = -457 | time = 2020-03-08T18:38:12.329
observer: state = -457 | time = 2020-03-08T18:38:12.329
```