1 Patróns de deseño OO

1.1 Sumario

1.2 Patróns OO

Deseñar software OO é difícil, e aínda máis difícil é deseñalo de xeito que sexa reusable. É habitual que novos programadores OO, abafados pola cantidade de opcións dispoñibles, non fagan bos deseños OO, utilizando técnicas non OO en programación OO.

Unha cousa que os programadores expertos saben é que non deben resolver os problemas partindo de cero, senón reusar solucións suficientemente probadas para casos similares. Así cando se atopa unha boa solución a un problema, esta empregarase unha e outra vez. Tal experiencia é o que fai un programador experto. Desta maneira atoparemos os mesmos patróns de clases en distinto software. Estes patróns resolverán problemas concretos facendo os deseños OO máis flexibles, elegantes e sobre todo, máis reutilizables. Un programador familiarizado con estes patróns aplicaraos inmediatamente sen ter que pensar como resolver problemas dos que xa existen boas solucións.

Os patróns de deseño dan nome, explican e avalían paradigmas concretos e recorrentes dentro da programación OO. O seu obxectivo é a captura de experiencias de deseño anteriores de xeito que os programadores as poidan empregar satisfactoriamente cando o problema se repite.

"Un patrón describe un problema que ocorre repetidamente e propón un modelo de solución para ser utilizala moitas veces de xeitos diferentes".

Un libro clave na teoría de patróns OO foi **Gang of Four-Design Patterns. Elements of Reusable Object-Oriented Software.** que categorizou os patróns xerais en POO describindo 23 deles agrupados en 3 categorías: **creacionais**, **estruturais** e **de comportamento**. Aínda que existen unha gran cantidade bibliografía sobre patróns, e infinidade de patróns OO, tanto xerais, como aplicados a ámbitos concretos, este libro segue a ser una referencia en canto ao deseño OO, e os patróns que nel se describen son amplamente utilizados.

A seguinte táboa mostra estes 23 patróns e as súas correspondentes categorías:

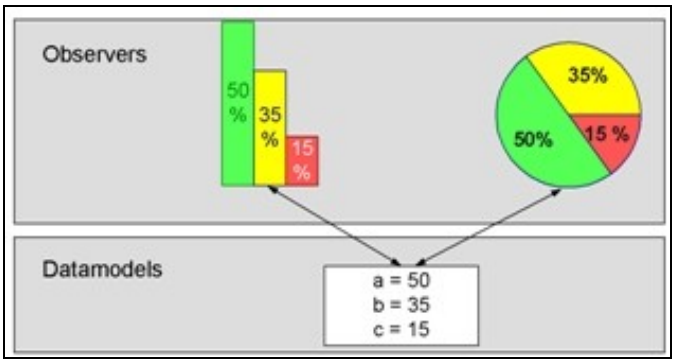
Categoría	Patróns
Creacional	Método factoría, Factoría abstracta, *Builder*, Prototipo, *Singleton*
Estructural	Adaptador, *Bridge*, *Composite*, *Decorador*, Fachada, *Proxy*
De comportamento	Interprete, Plantilla, Cadea de responsabilidade, *Command*, Iterador, *Mediator*, *Memento*, *Flyweight*, *Observer*, Estado, Estratexia, *Visitor*

Non veremos todos os patróns propostos neste libro, so faremos un breve percorrido por algúns dos máis sinxelos de xeito que alumno/a comprenda os fundamentos dun bo deseño OO.

É importante observar un erro típico de programadores novatos que tratan de utilizar nos seus deseños a maior cantidade posible de patróns, pensando que a maior número de patróns mellor deseño. Hai que ter en conta que o feito de utilizar un patrón non garante un bo deseño. Debemos, por tanto, comprender completamente os patróns que queremos implementar, e coñecer a que casos son aplicables e entender cales son as súas vantaxes.

1.2.1 Observer

Ás veces é preciso mostrar datos en máis dun formato determinado ao mesmo tempo, e que todas as vistas ou representacións se actualicen tamén ao mesmo tempo para reflectir os cambios que se poidan producir nos datos. Por exemplo, podemos querer representar os cambios nuns datos determinados que se representan en forma de diagrama de queixos e diagrama de barras simultaneamente. Cada vez que os datos cambian, é necesario que ambas as dúas representacións cambien sen ningún tipo de actualización pola nosa parte.



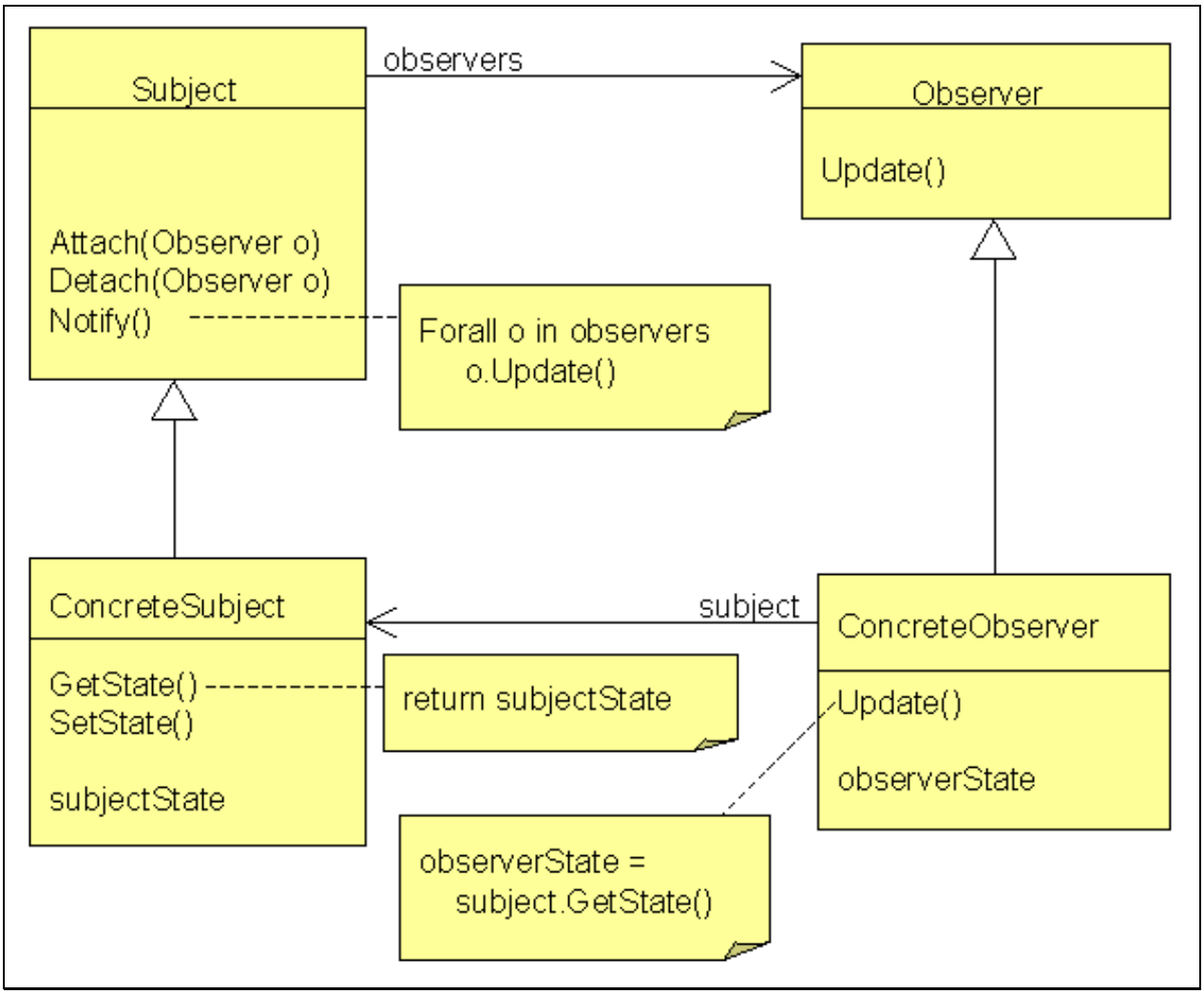
Outro exemplo de utilización deste patrón é o esquema **publicación-subscrición** onde cada vez que se produce un cambio nos datos do publicador se informa a todos os obxectos subscritos a esa publicación.

O patrón *Observer* describe o xeito de implementar este tipo de relacións. Os obxectos clave neste patrón son o *Subject* e mais o *Observer*. Cada *Subject* (xoga o rol do publicador) pode ter calquera número de *observers* dependentes (ou subscritos). Todos serán notificados cada vez que se produce un cambio no estado do *subject*.

Este patrón usarase nas seguintes situacións:

- Cando unha abstracción ten dous aspectos, un dependente do outro. Encapsulando estes aspectos en obxectos separados permitirá a súa modificación e reutilización dun xeito independente.
- Cando como resultado dun cambio nun obxecto, se require que outros cambien a súa vez e non sabemos o número destes.
- Cando un obxecto ten que notificar dun cambio no seu estado sen facer ningunha suposición acerca dos obxectos aos que lle ten que notificar o seu cambio. Dito doutro xeito, non queremos que eses obxectos estean acoplados.

1.2.1.1 Diagrama



1.2.1.2 Participantes

- **Subject**
  - Coñece os seus *observers*. Calquera número de *observers* pode observar este *subject*.
  - Proporciona un *interface* para engadir ou eliminar *observers* a este *subject*.

- **Observer**
  - Define un *interface* cun método *update* que será invocado cada vez que haxa que notificar ao *observer* dun cambio no estado *subject*.

- **ConcreteSubject**
  - Notifica todos os *observers* subscritos cando se produce un cambio no seu estado.
  - Almacenará o estado de interese para os *observers* subscritos a el.

- **ConcreteObserver**
  - Manterá un referencia ao *ConcreteSubject* que está a observar.
  - Almacenará un estado que debe ser consistente co estado do *subject*.
  - Implementará o *interface Observer* (o método *update*) o que lle permitirá manter o seu estado consistente co do Subject ao que está subscrito.

1.2.1.3 Exemplo

O seguinte exemplo mostra un *subject* chamado *ConcreteSubject* que mantén un lista de números enteiros. Os métodos que modifican o estado da lista de números serán engadir e eliminar un número da lista.

Sobre está lista crearemos dous *ConcreteObservers*, un deles mostrará a suma de todos os números da lista, mentres que o outro mostrará todos os elementos da lista.

```
public interface Subject {
```

```
public void addObserver(Observer obs);
public void removeObserver(Observer obs);
}
```

Este será o interface que deberán implementar todos os *subjects* concretos. Neste caso non aparece o método *notify*, tal e como viches no diagrama de clases. Este método pode aparecer opcionalmente no interface ou non, en función de se se implementa nos *subjects* concretos como público (aparecerá no interface) ou como privado.

```
public interface Observer {
public void update(Subject o);
}
```

O interface *Observer* conterá únicamente un método chamado *update*. Cada *observer* concreto debe implementar este método co código concreto que permita a actualización de ese *observer* a partires dos cambios que se produzan no *subject*

```
import java.util.ArrayList;
import java.util.Iterator;
public class IntegerDataBag implements Subject {
	private ArrayList<Integer> list = new ArrayList<Integer>();
	private ArrayList<Observer> observers = new ArrayList<Observer>();

	public void add(Integer i) {
		list.add(i);
		notifyObservers();
	}
	public Iterator<Integer> iterator() {
		return list.iterator();
	}
	public Integer remove(int index) {
		if(index < list.size()) {
			Integer i = (Integer) list.remove(index);
			notifyObservers();
			return i;
		}
		return null;
	}
	public void addObserver(Observer o) {
		observers.add(o);
	}
	public void removeObserver(Observer o) {
		observers.remove(o);
	}
	private void notifyObservers() {
		// Bucle a través de todos os observers para enviarlles a notificacion
		Iterator<Observer> i = observers.iterator();
		while(i.hasNext()) {
			Observer o = (Observer) i.next();
			o.update(this);
		}
	}
}
```

A clase *IntegerDataBag* conterá unha implementación do interface *Subject*. Nela manterase unha lista cos *observers* que a están observando(neste caso dentro dun *ArrayList*), de xeito que cando se produza un cambio no seu estado (a través dos métodos *add(Integer i)* e *remove(int index)*) se notifique a todos eles. O método *notifyObservers()* é o encargado de realizar esa notificación.

```
import java.util.Iterator;
public class IntegerPrinter implements Observer {
	private IntegerDataBag bag;

	public IntegerPrinter(IntegerDataBag bag) {
		this.bag = bag;
		bag.addObserver(this);
	}
	public void update(Subject o) {
		if( o == bag ) {
			System.out.println( "IMPRESOR:O contido da bolsa de enteiros cambiou." );
			System.out.println( "IMPRESOR:O bolsa de enteiros conten o seguinte:" );
			Iterator<Integer> i = bag.iterator();
			System.out.print("IMPRESOR:");
			while( i.hasNext() ) {
```

```
                        System.out.print(i.next() + " ");
                    }
                    System.out.println();
                }
        }
}
```

A clase *IntegerPrinter* será un *observer* concreto, encargado de mostrar por consola todos os elementos dentro da lista de enteiros. Cada vez que a lista de enteiros cambia ten que volver a escribirse na consola a lista. O método *update* (o que o interface *Observer* obriga a implementar) é o encargado de escribir esa lista.

```
import java.util.Iterator;

public class IntegerAdder implements Observer {

    private IntegerDataBag bag;
    public IntegerAdder(IntegerDataBag bag) {
this.bag = bag;
bag.addObserver(this);
    }
    public void update(Subject o) {
        if( o == bag ) {
    System.out.println("SUMADOR:O contido da bolsa de Enteiros cambiou.");
    int suma = 0;
    Iterator<Integer> i = bag.iterator();
    while( i.hasNext() ) {
Integer integer = ( Integer ) i.next();
suma+=integer.intValue();
    }
    System.out.println("SUMADOR:A suma de todos os integers é: " + suma);
        }
    }
}
```

A clase *IntegerAdder* será outro *observer* sobre a lista de enteiros. Neste caso mostrarase por consola a suma de todos os elementos que forman a bolsa de enteiros. Cada vez que cambie a bolsa (porque se engada ou elimine un número) obviamente a súa suma cambiará e por tanto debe reescribirse na consola.

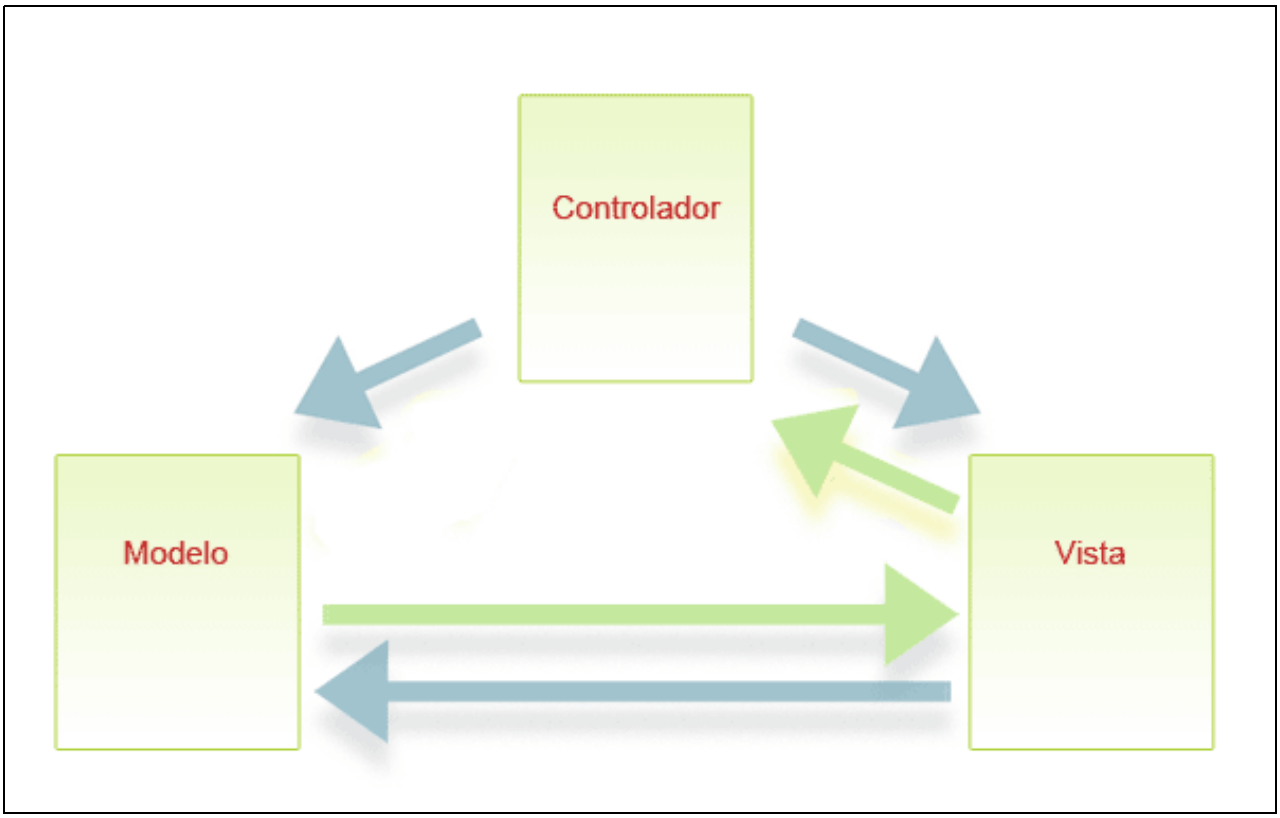
```
public class Principal {
      public static void main( String [] args ) {
            Integer i1 = new Integer(1);
            Integer i2 = new Integer(2);
            Integer i3 = new Integer(3);
            Integer i4 = new Integer(4);
            Integer i5 = new Integer(5);
            Integer i6 = new Integer(6);
            Integer i7 = new Integer(7);
            Integer i8 = new Integer(8);
            Integer i9 = new Integer(9);
            IntegerDataBag bolsa = new IntegerDataBag();
            bolsa.add(i1);
            bolsa.add(i2);
            bolsa.add(i3);
            bolsa.add(i4);
            bolsa.add(i5);
            bolsa.add(i6);
            bolsa.add(i7);
            bolsa.add(i8);
            IntegerAdder sumador = new IntegerAdder(bolsa);
            IntegerPrinter impresor = new IntegerPrinter(bolsa);

            // Sumador e impresor son 2 observers engadidos a bolsa
            System.out.println( "Antes de engadir outro enteiro a bolsa:" );
            bolsa.add(i9);
            System.out.println();
            System.out.println("Antes de borrar un integer da bolsa:");
            bolsa.remove(0);
      }
}
```

Este é a clase conductora, encargada de poñer en funcionamento o patrón.

1.2.2 MVC

O MVC(Modelo-Vista-Controlador) está formado por tres tipos de obxectos. O **modelo** é o obxecto da aplicación, a **vista** é a pantalla de presentación e o **controlador** describe o xeito en que a vista se conecta co modelo.

1.2.2.1 Diagrama



1.2.2.2 Participantes

- **Modelo**: representa os datos e as regras de negocio que os gobernan e modifican. Normalmente o modelo é unha aproximación ao proceso no mundo real.
- **Vista**: mostra os datos contidos no modelo. Accede os datos a través do modelo e di como eses datos deben ser presentados. É responsabilidade da vista manter a consistencia co modelo, de xeito que se esta cambia, a vista tamén debe cambiar. Isto pode acadarse mediante modelos activos (cada vez que o modelo cambia avisa a todas as vistas rexistradas con el) o con modelos pasivos (son as vistas as que o interrogan sobre algún posible cambio).
- **Controlador**: mapea as interaccións que se producen sobre a vista coas accións que realiza o modelo. Por exemplo, nun cliente *Swing*, as interaccións son os clicks sobre botóns, seleccións en listas, etc que o usuario fai sobre o interface, mentres que nunha aplicación web son as peticións *GET* ou *POST HTTP*.

MVC desacopla as vistas do modelo establecendo un protocolo de subscrición/notificación entre eles. Unha vista debe asegurar que nela se reflicte o estado do modelo. Sempre que o modelo cambie, este debe notificar as vistas que depende del. Como resposta ao cambio no modelo, a vista terá que actualizarse a si mesma.

Esta aproximación permitirá acoplar múltiples vistas a un único modelo para mostrar diferentes presentacións. Tamén podemos crear novas vistas para un modelo sen volver a reescribilo.

Posto que as vistas son independentes do modelo, podemos tamén modificar unhas ou o outro dun xeito independente. Isto é posible debido a introdución de este compoñente a maiores que é o controlador.

O MVC e unha especialización do modelo *Observer*, no cal non aparecen obxectos de tipo *controller*. MVC ten comunmente outras características: as vistas pódense aniñar de xeito que unha vista complexa estea formada por varias vistas simples; unha vista utilizará unha instancia concreta dun controlador para implementar unha estratexia particular de resposta (para cambiar a estratexia so temos que utilizar un tipo diferente de controlador).

En definitiva, podemos considerar MVC como un patrón complexo que permite desacoplar os modelos de datos das súas vistas. Ten presente que o patrón MVC pode usar outros patróns (o patrón *Composite* para implementar vistas aniñadas, o patrón Estratexia para permitir diferentes controladores

con diferentes respotas, o patrón Decorador para engadir scolling as vistas , etc).

Son os eventos os que normalmente orixinan que o controlador modifique o modelo ou a vista. Cando un controlador modifica os datos ou propiedades dun modelo, todas as vistas sobre o mesmo deben ser actualizadas. De xeito similar cando o controlador modifica unha vista, esta debe actualizarse de novo cos datos do seu modelo.

1.2.2.3 Exemplo

O seguinte exemplo mostra unha sinxela calculadora (so multiplica e pon a 1) utilizando o patrón MVC:

```
import java.math.BigInteger;
public class CalcModel {
    //... Constante
    private static final String VALOR_INICIAL = "1";
    private BigInteger numero;
    public CalcModel() {
        reset();
    }
    public void reset() {
        numero = new BigInteger(VALOR_INICIAL);
    }
    public void multiplyBy(String operand) {
        numero = numero.multiply(new BigInteger(operand));
    }
    public void setValue(String value) {
        numero = new BigInteger(value);
    }
    public String getValue() {
        return numero.toString();
    }
}
```

O **modelo** conterá os datos, neste caso unicamente un *BigInteger* co resultado da operación da calculadora, e as regras ou métodos que neste caso serán unicamente multiplicar e resetear. Observa ademais que aparece un *setter* e un *getter* para *numero*.

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class CalcView extends JFrame {
    private static final String VALOR_INICIAL = "1";
    private JTextField m_userInputTf = new JTextField(5);
    private JTextField m_totalTf     = new JTextField(20);
    private JButton    m_multiplyBtn = new JButton("Multiply");
    private JButton    m_clearBtn    = new JButton("Clear");
    public CalcView() {
        reset();
        m_totalTf.setEditable(false);
        JPanel content = new JPanel();
        content.setLayout(new FlowLayout());
        content.add(new JLabel("Input"));
        content.add(m_userInputTf);
        content.add(m_multiplyBtn);
        content.add(new JLabel("Total"));
        content.add(m_totalTf);
        content.add(m_clearBtn);
        this.setContentPane(content);
        this.pack();
        this.setTitle("Calculadora simple - MVC");
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setVisible(true);
    }
    void reset() {
        m_totalTf.setText(VALOR_INICIAL);
    }
    String getUserInput() {
        return m_userInputTf.getText();
    }
    void setTotal(String newTotal) {
        m_totalTf.setText(newTotal);
    }
    void showError(String errMessage) {
```

```
        JOptionPane.showMessageDialog(this, errMessage);
    }
    void addMultiplyListener(ActionListener mal) {
        m_multiplyBtn.addActionListener(mal);
    }
    void addClearListener(ActionListener cal) {
        m_clearBtn.addActionListener(cal);
    }
}
```

A **vista** será unha ventá *Swing* onde se creara un interface de usuario para esa calculadora simple. Observa que existen getter e setters para os *JTextField* permitindo deste xeito modificar ou obter o seu contido. Os métodos *addMultiplyListener(ActionListener mal)* e *addClearListener(ActionListener cal)* serán os encargados de engadir os *Listeners* axeitados para os dous botóns presentes no interface.

```
import java.awt.event.*;
public class CalcController {
    private CalcModel m_model;
    private CalcView  m_view;

    CalcController(CalcModel model, CalcView view) {
        m_model = model;
        m_view  = view;
        // Engadir os listeners a vista
        view.addMultiplyListener(new MultiplyListener());
        view.addClearListener(new ClearListener());
    }
    class MultiplyListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String userInput = "";
            try {
                userInput = m_view.getUserInput();
                m_model.multiplyBy(userInput);
                m_view.setTotal(m_model.getValue());

            } catch (NumberFormatException nfex) {
                m_view.showError("Bad input: '" + userInput + "'");
            }
        }
    }
    class ClearListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            m_model.reset();
            m_view.reset();
        }
    }
}
```

O **controlador** é o encargado de traducir as accións do usuario que se producen sobre a vista en accións no modelo. Así, o premer o botón multiplicar na vista orixina que se faga unha chamada ao modelo indicándolle que debe multiplicar numero por o valor introducido no *JTextField* e escribir o seu resultado de novo na vista. Observa que o único que fai este controlador é engadir á vista os listeners que nel se definen. En cada un destes listeners é onde se indica as accións do modelo resposta aos eventos sobre a vista.

```
public class CalcMVC {
    public static void main(String[] args) {
        CalcModel      model      = new CalcModel();
        CalcView       view       = new CalcView();
        CalcController controller = new CalcController(model, view);
    }
}
```

Por último temos a clase principal ou condutora onde se crean as restantes clases.

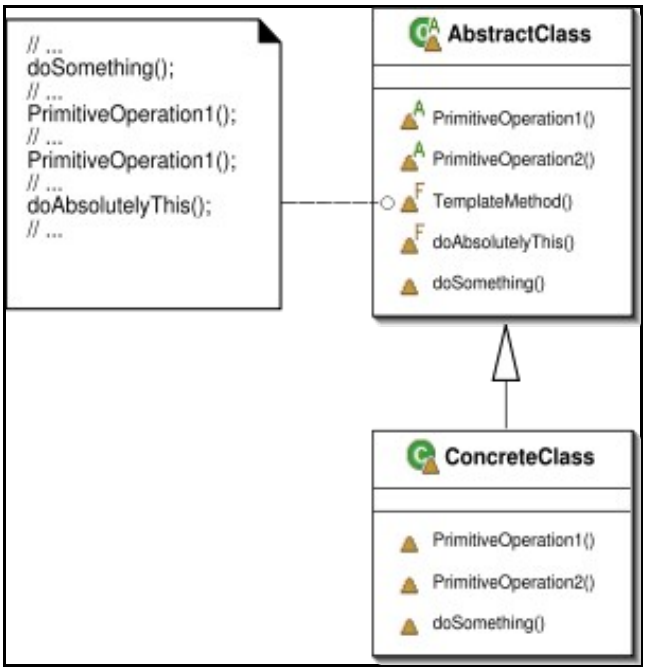
1.2.3 *Templates*

Definen o esqueleto dun algoritmo, deixando algúns pasos para ser implementados polas subclases. Os métodos *template* deixan ás subclases redefinir certos pasos dun algoritmo sen cambiar a súa estrutura. Así, un método *template* define un algoritmo en termos de métodos abstractos que as subclases implementarán. O método *template* debe usarse:

- Para implementar as partes invariantes dun algoritmo e deixar para as subclases as partes que poden variar.

- Cando certas partes son comúns a todas as subclases, de xeito que as localicemos nunha única superclase da que herdarán estas subclases, evitando así duplicación de código.
- Para controlar as extensións que as subclases fan dun algoritmo. As subclases terán necesariamente que sobreescribir os métodos abstractos, non poderán sobreescribir os métodos finais, e poderán opcionalmente sobreescribir os métodos *hook*

1.2.3.1 Diagrama



1.2.3.2 Participantes

- **Clase abstracta**
  - ♦ define as métodos primitivos ou abstractos que implementarán as subclases.
  - ♦ Implementa os métodos *template* que definen o esqueleto dun algoritmo. Un método *template* chama outros métodos primitivos ou abstractos así como outros métodos non abstractos que poidan pertencer a clase abstracta.
- **Clase concreta**
  - ♦ Implementa os métodos primitivos ou abstractos declarados na súa superclase. Opcionalmente pode sobreescribir algúns outros métodos aos que a superclase dotou dun comportamento por defecto (chamarémoslles métodos *hook*).

1.2.3.3 Exemplo

```
public abstract class Xogo {
     private int contadorXogadores;
     public abstract void inicializarXogo();
     public abstract void xogar(int xogador);
     public abstract boolean finDoXogo();
     public abstract void imprimirGañador();

     /* O método "template" : */
     public final void xogarUnhaPartida(int contadorXogadores) {
         this.contadorXogadores= contadorXogadores;
         inicializarXogo();
         int j = 0;
         while (!finDoXogo()){
             xogar(j);
             j = (j + 1) % contadorXogadores;
         }
         imprimirGañador();
     }
 }
```

A clase abstracta será común a varios xogos diferentes, nos que uns xogadores se enfrontan a outros, xogando un cada turno.

```
public class Monopoly extends Xogo{
     /* Implementación (obrigatoria) dos métodos abstractos */
     public void inicializarXogo() {
```

```
            // ...
        }
        public void xogar(int Xogador) {
            // ...
        }
        public boolean finDoXogo() {
            // ...
        }
        public void imprimirGañador() {
            // ...
        }
        /* A partir de aqui aparecerán os métodos específicos do Monopoly. */
        // ...
    }
```

Esta clase concreta terá os métodos específicos propios do Monopoly, tanto as implementacións dos métodos abstractos definidos pola superclase, como aqueles que lle serán propios.

```
public class Xedrez extends Xogo{
        /* Implementación (obrigatoria) dos métodos abstractos */
        public void inicializarXogo() {
            // ...
        }
        public void xogar(int Xogador) {
            // ...
        }
        public boolean finDoXogo() {
            // ...
        }
        public void imprimirGañador() {
            // ...
        }
        /* A partir de aqui aparecerán os métodos específicos do Xedrez. */
        // ...
    }
```

A clase concreta Xedrez conterá unha implementación diferente dos métodos abstractos definidos para na superclase con respecto a clase Monopoly. Por tanto cada un deles contén algúns detalles concretos dependentes do xogo para un algoritmo que permanece común para todos os xogos.