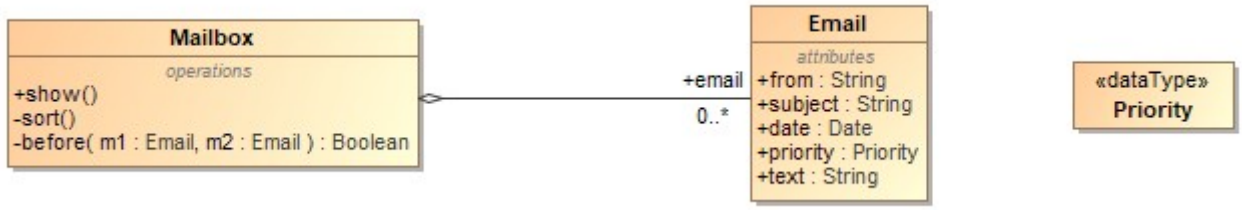
Ingeniería del Software Avanzada

Práctica 1 Patrones de Diseño



Autor: David Ramírez Arco

Fecha: 1 de mayo de 2022

GitHub: https://github.com/Archerd6/Practica-1--design-pattern

Índice

Cuestiones

Cuestión 1

Consideremos los siguientes patrones de diseño: Adaptador, Decorador y Representante. Identifique las principales semejanzas y diferencias entre cada dos ellos (no es suficiente con definir, sino describir explícitamente similitudes y semejanzas concretas).

El patrón Adaptador consiste en la implementación de una clase que permita reutilizar clases adaptándolas al funcionamiento otras nuevas.

El patrón Decorador permite extender comportamientos de clases de forma dinámica, para convertir una clase previamente desarrollada en una con un comportamiento diferente, sin la necesidad de crear una clase nueva o subclase de esta.

El patrón Representante proporciona un nivel adicional para proporcionar acceso controlado o inteligente (a través de un proxy).

Por tanto, la principal semejanza es que los tres tienen un propósito estructural, aunque cada uno soluciona un problema distinto, en posibles situaciones distintas.

- Una similitud entre el patrón adaptador y el decorador modifican objetos ya existentes
  (El patrón adaptador adapta una interfaz para poder reutilizar una clase existente y el decorador añade funciones a objetos mediante composición)

- Y una diferencia por ejemplo, el patrón representante no busca transformar directamente el comportamiento de ninguna clase.

Cuestión 2

Consideremos los patrones de diseño de comportamiento Estrategia y Estado. Identifique las principales semejanzas y diferencias entre ellos.

La principal similitud es que los dos modifican el comportamiento de los objetos: Ambos buscan aportar diversas formas de funcionamiento a una clase.

La diferencia entre ellos es que en el patrón estado otorga diferentes funcionalidades dadas las circunstancias en las que se encuentre una clase, mientras que en el patrón estrategia desarrolla nuevas formas de trabajar mediante la elección de las circunstancias.

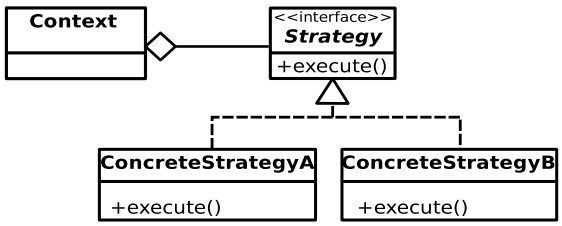


Figura 1: Patrón Estrategia

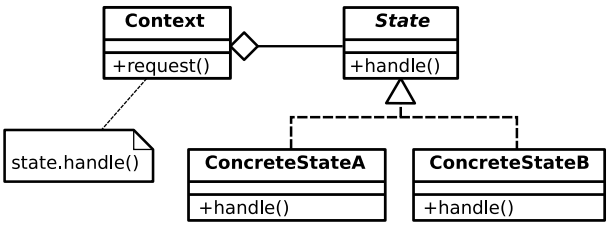


Figura 2: Patrón Estado

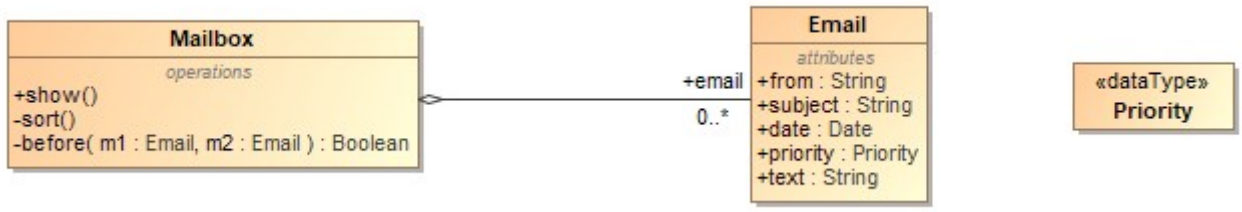
Cuestión 3

Consideremos los patrones de diseño de comportamiento Mediador y Observador. Identifique las principales semejanzas y diferencias entre ellos.

La principal similitud entre los patrones es que son de comportamiento, al igual que los anteriores modifican las relaciones entre los objetos. Ambos son bastante similares ya que pretenden conectar objetos emisores y receptores de información.

El patrón mediador crea un objeto que define como los objeto interactuaran entre sí. El patrón observador no crea un objeto, si no que define objetos como Observador u Observables

Cliente de un correo e-look




```
private void sort() {
   for ( int i = 2; i <= email.size(), i++ )
       for ( int j = email.size(); j >= i; j-- )
           if ( before(email.at(j),email.at(j-1)) )
               // intercambiar los mensajes j y j-1
               ...
   }
```

Identifique un patrón de diseño que nos permita resolver la situación presentada, permitiendo incluso cambiar de un criterio de ordenación a otro mientras el usuario utiliza e-look, y de forma que sea fácilmente extensible (por ejemplo, para ordenar por otros criterios aparte de los indicados). Mostrar de forma esquemática la implementación en Java del patrón propuesto al problema descrito

La clase Mailbox es la que contiene y controla todos los correos que le llegan al cliente. Los muestra utilizando el método show(), que a su vez utiliza función sort() para ordenar los correos.

Al tener varias estrategias para hacer lo mismo, la mejor elección sería usar el patrón estrategia para así definir los distintos criterios de ordenación (responsabilidad del método sort())

Para utilizar este patrón de diseño crearemos la interfaz Estrategia, que tendrá el método `before(Email m1, Email m2)` que será implementado según el criterio que el cliente elija.

Implementación en java

El proyecto tiene la siguiente estructura:

El cliente guardará sus `Email` en `Mailbox` (concretamente en un `ArrayList`) ordenándolos según el criterio (`Estrategia`) que él elija, el cual podrá cambiar en cualquier momento.

• Clase `Mailbox`: En esta clase se define un Mailbox, objeto que usará e-look para almacenar los emails

```java
package e_look;

import java.util.ArrayList;
import java.util.Iterator;

public class Mailbox
{
	ArrayList<Email> mails;
	Estrategia orden_a_seguir;

	public Mailbox(Estrategia orden_inicial)
	{
		mails = new ArrayList<Email>();
		orden_a_seguir=orden_inicial;
	}

	/**Muestra por terminal los mails*/
	public void show()
	{
		sort();
		Iterator<Email> iter = mails.iterator();
		while(iter.hasNext())
		{
			String email_x = iter.next().toString();
			System.out.println(email_x);
		}
	}

	/**Usa el algoritmo de ordenacion bubble sort*/
	private void sort()
	{
		for (int i = 0; i < mails.size()-1; i++)
	    for (int j = 0; j < mails.size()-i - 1; j++)
				if ( this.before(mails.get(j),mails.get(j+1)) )
				{
					// intercambiar los mensajes j y j-1
					Email temp = mails.get(j);
					mails.set(j, mails.get(j+1));
					mails.set(j+1, temp);

				}


	}

	/**¿Esta antes el email m1 que el m2?*/
	private boolean before(Email m1, Email m2)
	{
		return orden_a_seguir.before(m1, m2);
	}

}
```

• Clase `Email`: Esta clase representa los emails

```java
package e_look;

import java.util.Date;

public class Email
{
	String from;
	String subject;
	String text;
	Date date;
	Priority priority;

	public Email(String from1, String subject1,String text1,Date date1,Priority priority1)
	{
		from=from1;
		subject=subject1;
		text=text1;
		date=date1;
		priority=priority1;
	}

	@Override
	public String toString()
	{
		return "e-look Mail info (From: "+this.from+", Subject: "+this.subject+", Text: "+this.text+", Date: "+this.date.toString()+", Priority: "+this.priority+")";
	}
}
```

• Enumerado `Priority`: Representa las posibles prioridades

```java
package e_look;

public enum Priority
{
	BAJA,MEDIA,ALTA
}
```

• Interfaz `Estrategia`: Esta interfaz es la que se encargar de ser el molde para las distintas estrategias

```java
package e_look;

public interface Estrategia
{
	boolean before(Email e1, Email e2);
}
```

Y las clases que implementan esta interfaz

```java
package e_look;

public class Estrategia_Date implements Estrategia
{
	public boolean before(Email m1, Email m2)
	{
		if(m1.date.compareTo(m2.date) < 0)
		{
			return false;
		}
		return true;
	}
}
```

```java
package e_look;

public class Estrategia_From implements Estrategia
{
	public boolean before(Email m1, Email m2)
	{
		if(m1.from.compareToIgnoreCase(m2.from) < 0)
		{
			return true;
		}
		return false;
	}
}
```

```java
package e_look;

public class Estrategia_Priority implements Estrategia
{
	public boolean before(Email m1, Email m2)
	{
		if(m1.priority.compareTo(m2.priority) < 0)
		{
			return true;
		}
		return false;
	}
}
```

```java
package e_look;

public class Estrategia_Subject implements Estrategia
{
	public boolean before(Email m1, Email m2)
	{
		if(m1.subject.compareToIgnoreCase(m2.subject) < 0)
		{
			return true;
		}
		return false;
	}
}
```

```java
package e_look;

public class Estrategia_Text implements Estrategia
{
	public boolean before(Email m1, Email m2)
	{
		if(m1.text.compareToIgnoreCase(m2.text) < 0)
		{
			return true;
		}
		return false;
	}
}
```

◈ Y como clase de pruebas he creado `Test_e_look`:

```java
package e_look;

import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Test_e_look
{
	public static void main(String[] args)
	{
		Mailbox e_look_Mailbox=new Mailbox(new Estrategia_Priority());

		Date primera_fecha = null;
		Date segunda_fecha = null;
		Date tercera_fecha = null;
		try
		{
			primera_fecha = new SimpleDateFormat("yyyy-MM-dd").parse("2022-06-01");
			segunda_fecha = new SimpleDateFormat("yyyy-MM-dd").parse("2022-06-02");
			tercera_fecha = new SimpleDateFormat("yyyy-MM-dd").parse("2022-06-03");
		}
		catch (ParseException e)
		{
			e.printStackTrace();
		}

		Email primer_email=new Email("Manu","Planes","¿Que vas a hacer mañana para tu cumple?",primera_fecha,Priority.BAJA);
		Email segundo_email=new Email("Sergio","Esperadme","Que hay atasco y llego un poco tarde",segunda_fecha,Priority.ALTA);
		Email tercer_email=new Email("Miguel","Review","La fiesta fue perfect :)",tercera_fecha,Priority.MEDIA);

		e_look_Mailbox.mails.add(primer_email);
		e_look_Mailbox.mails.add(segundo_email);
		e_look_Mailbox.mails.add(tercer_email);

		e_look_Mailbox.show();

//		e_look_Mailbox.Orden = new Estrategia_Date();
	}

}
```

Conclusión

El patrón estrategia se adapta muy bien para este proyecto (ya que permite la modificación y la implementación de los criterios de ordenación de una forma cómoda)