UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA – ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
71022011 – DISEÑO DE APLICACIONES ORIENTADA A OBJETOS
(GRADO EN INGENIERÍA INFORMÁTICA / TECNOLOGÍAS DE LA INFORMACIÓN)
JUNIO 2013 – MODELO B– NO ESTÁ PERMITIDO EL USO DE MATERIAL ADICIONAL

PARTE TEÓRICA - TEST [2,5 PUNTOS]:

Sólo una de las respuestas es válida. Las respuestas correctas se puntuarán con +1.0, mientras que las respondidas de manera incorrecta se puntuarán con -0.25. Las no contestadas no tendrán influencia ni positiva ni negativa en la nota.

Pregunta 1: ¿Cuál es el objetivo del patrón Strategy? (p 289)

a) Definir el esqueleto de un algoritmo dejando la implementación de algunos de los pasos del esqueleto a las subclases.
b) Definir una familia de algoritmos, encapsular cada uno de ellos y hacerlos intercambiables. Permite que un algoritmo varíe independientemente de los clientes que lo usan.
c) Permite añadir dinámicamente nuevas responsabilidades/nuevo comportamiento a un objeto, proporcionando una alternativa a la definición de subclases
d) Facilitar la construcción de objetos relacionados entre sí.

Pregunta 2: ¿Cuál de las siguientes afirmaciones acerca del patrón State es falsa? (p 279-281)
a) Se aplica cuando el comportamiento de un objeto depende de su estado y debe cambiar en tiempo de ejecución.
b) Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados.
c) Los objetos estado nunca pueden compartirse.
d) Hace explícitas las transiciones entre estados.

Pregunta 3: ¿En que ocasiones no es recomendable aplicar el patrón Flyweight? (p 181)
a) Los costes de almacenamiento son elevados debido a la gran cantidad de objetos creados.
b) La mayor parte del estado del objeto puede hacerse extrínseco.
c) Muchos grupos de objetos pueden reemplazarse por relativamente pocos objetos compartidos.
d) La aplicación depende de la identidad de un objeto.

Pregunta 4: ¿Qué patrón aplicaría en el caso en el que una clase principal delegue en sus subclases el tipo de objetos a crear? (p 100)
a) Abstract Factory
b) Factory Method
c) Decorator
d) Builder

Pregunta 5: ¿Cuál de las siguientes afirmaciones acerca del patrón Chain of Responsibility es falsa? (p. 207)
a) Desacopla el objeto que envía la petición de sus receptores.
b) Simplifica el objeto al no ser necesario que éste conozca la estructura de la cadena y que mantenga una referencia directa a cada uno de sus elementos.
c) Facilita la creación o eliminación de responsabilidades de manera dinámica.
d) Es necesario especificar explícitamente el receptor al que va dirigida la petición.

Pregunta 6: ¿Qué patrón aplicaría en el caso de querer definir una nueva operación sin cambiar las clases de los elementos sobre los que opera? (p. 305)
a) State
b) Visitor
c) Memento
d) Observer

Pregunta 7: Dado el siguiente fragmento de código, ¿Cómo podríamos invocar el constructor de la clase Base para que mostrara en consola el mensaje "constructor base"?

```
class Base{
    Base(int i){
          System.out.println("constructor base");
    }
    Base(){
    }
}

public class Sup extends Base{
    public static void main(String argv[]){
          Sup s= new Sup();
          //POSICION 1
    }

    Sup()
    {
          //POSCION 2
    }
     public void derived()
    {
          //POSICION 3
    }
}
```

a) Justo después de la línea //POSICION 1, escribiendo Base(10).
b) Justo después de la línea //POSICION 1, escribiendo super(10).
c) Justo después de la línea //POSICION 2, escribiendo super(10).
d) Justo después de la línea //POSICION 3, escribiendo super(10).

Pregunta 8: ¿Qué modificadores serían válidos para la clase MiClaseInterna?

```
public class MiClase {

public static void main(String argv[]){ }
/*Modificador AQUI*/ class MiClaseInterna {}

}
```

a) sólo public
b) sólo private
c) sólo static
d) public, private y static

Pregunta 9: Dado el siguiente código, ¿cuál de las afirmaciones es cierta?

Número de Línea Código

```
4               public class ClaseA extends ClaseB {
5                  public static void main(String[] args) {
6                    Short cuenta = 7;
7                    System.out.println(contar (cuenta, 6));
8                  }
9               }
10              class ClaseB {
11                  int contar (int x, int y) { return x + y; }
12              }
```

a) 13
b) Fallo de compilación debido a más de un error.
c) Error de compilación en la línea 6.
d) Error de compilación en la línea 7.

Pregunta 10: ¿Qué sentencia deberíamos insertar en el siguiente código para que éste compile y se ejecute correctamente?

```
class ClaseA { }
class ClaseB extends ClaseA { }
class ClaseC extends ClaseB { }
public class ClaseD {
  ArrayList<ClaseB> ir() {
  / INSERTAR EL CODIGO AQUÍ
  }
}
```

a) return new ArrayList<ClaseC>();
b) return new ArrayList<Object>();
c) return new ArrayList<ClaseA>();
d) Ninguna de las anteriores

PARTE PRÁCTICA [6,5 PUNTOS]:

Pregunta 11: Suponga una biblioteca de clases donde existe una función Add que toma como parámetros dos enteros y devuelve la suma de ambos. Suponga ahora que se ha modificado el método Add de la biblioteca, de modo que ahora recibe como parámetros dos valores decimales. ¿Se podría utilizar el patrón Adapter de modo que no fuera necesario modificar el código de la clase cliente que hacía uso del método Add inicial?

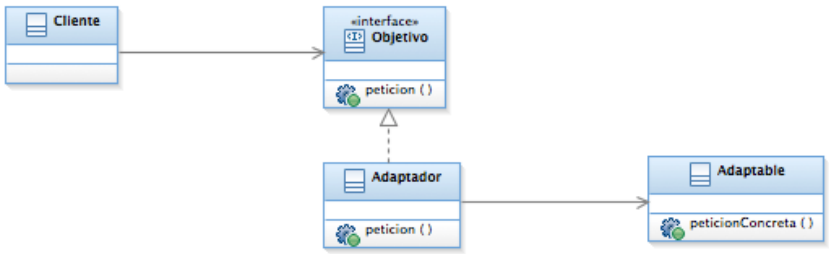
a) **[1 punto]** Justifique la elección de este patrón y analiza las ventajas y desventajas de la elección. ¿Hay otro patrón que se podría haber usado en lugar de este?
b) **[1 punto]** Plantee el diagrama de clases necesario para abordar la implementación problema propuesto. Comience con un planteamiento genérico y luego particularícelo al problema concreto. Explique detalladamente todas las decisiones tomadas.
c) **[1,25 puntos]** Implemente en Java el diagrama de clases propuesto.

Apartado a

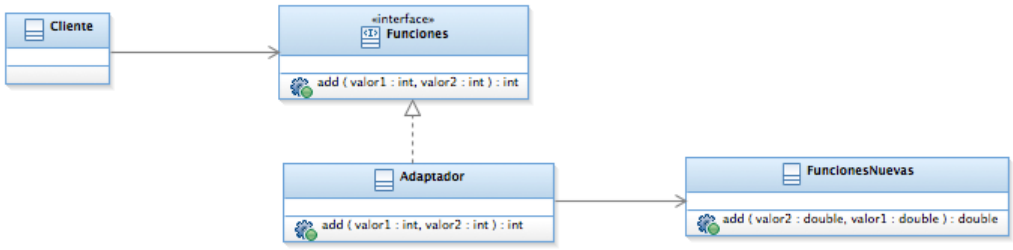
El patrón Adapter convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes. Se puede encontrar más información en la página 131 del libro base de la asignatura. Hay una interesante explicación de este patrón en la página 236 del libro Head First Design Patterns de O'Reilly. Este patrón resulta la opción más adecuada para resolver el problema planteado.

Apartado b

El esquema general del patrón es el siguiente:



Su adaptación al problema planteado resulta muy sencilla y se puede entender a través del siguiente diagrama de clases, donde puede verse que hemos declarado una interfaz Funciones, que va a ser implementada por la clase Adaptador. La función add de la clase Adaptador es la encargada de realizar la llamada al método add de la clase FuncionesNuevas, abstrayendo al cliente de la nueva configuración del método add.



Apartado c

La implementación del patrón sería la siguiente:

- **Interfaz Funciones:**

```
package edu.uned.codigo;

public interface Funciones {

	public int add(int valor1, int valor2);
}
```

- **Clase a adaptar FuncionesNuevas:**

```
package edu.uned.codigo;

public class FuncionesNuevas {

	public double add(double valor2, double valor1) {
		return valor1+valor2;
	}
}
```

- **Clase Adaptador:**

```
package edu.uned.codigo;

public class Adaptador implements Funciones {

	private FuncionesNuevas funcionesNuevas=new FuncionesNuevas();

	public int add(int valor1, int valor2) {
		return (int) funcionesNuevas.add(valor1, valor2);
	}
}
```

- **Clase Cliente:**

```
package edu.uned.codigo;

public class Cliente {

	private static Funciones funciones;

	public static void main(String[] args) {

		int a,b;

		funciones=new Adaptador();

		a=10;
		b=20;

		System.out.println("El resultado de sumar los dos enteros es: "
			+funciones.add(a, b));
	}
}
```

Pregunta 12: ¿Se podría usar el patrón Observer para un sistema de gestión de Temperaturas que sea capaz de actualizar un conjunto de dispositivos con la temperatura detectada por una estación central de Temperaturas?

a) **[1 punto]** Justifique la elección de este patrón y analiza las ventajas y desventajas de la elección. ¿Hay otro patrón que se podría haber usado en lugar de este?
b) **[1 punto]** Plantee el diagrama de clases necesario para abordar la implementación problema propuesto. Comience con un planteamiento genérico y luego particularícelo al problema concreto. Explique detalladamente todas las decisiones tomadas.
c) **[1,25 puntos]** Implemente en Java el diagrama de clases propuesto.

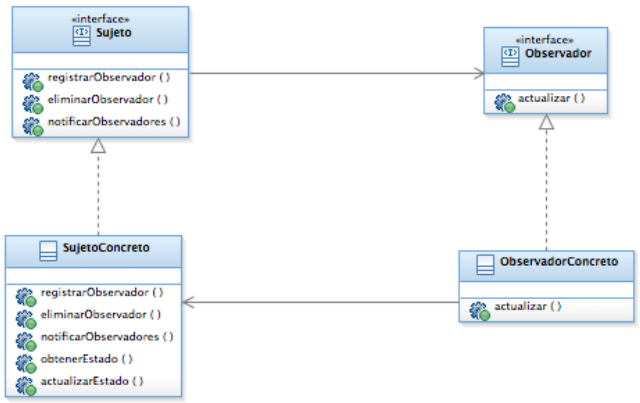
Apartado a

El patrón Observador define una dependencia uno a muchos entre objetos, de forma que cuando un objeto cambia su estado (el objeto observado), se procede a notificar al conjunto de objetos que están observando de modo que éstos puedan reaccionar de algún modo, por ejemplo actualizando su estado.

Se puede encontrar más información en la página 269 del libro base de la asignatura. Hay una interesante explicación de este patrón en capítulo 2 del libro Head First Design Patterns de O'Reilly. Este patrón resulta la opción más adecuada para resolver el problema planteado.

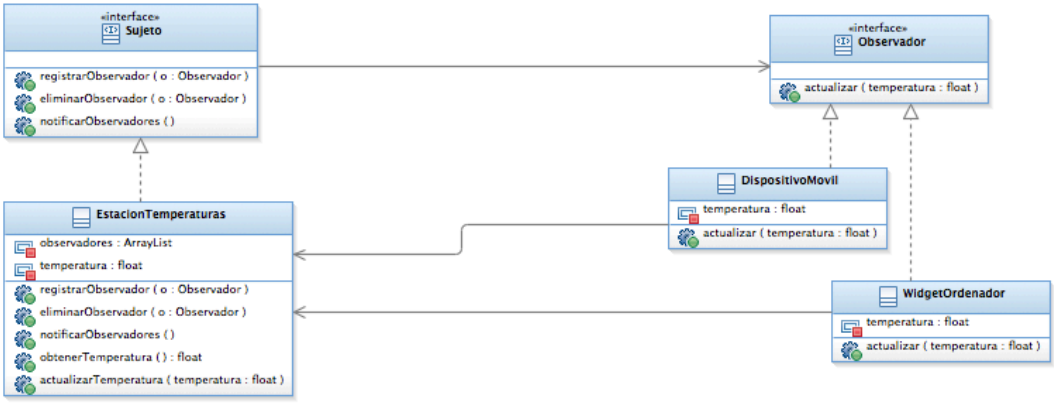
Apartado b

El esquema general del patrón es el siguiente:



Su adaptación al problema planteado resulta muy sencilla y se puede entender a través del siguiente diagrama de clases, donde puede verse que hemos construido sobre la interfaz Sujeto la clase EstacionTemperaturas. Esta clase es la encargada de tomar las temperaturas cada cierto tiempo y actualizar su estado. Este estado está reflejado en la variable de instancia temperatura de la clase. Nótese como, con el fin de poder almacenar el conjunto de objetos que van a "observar" la estación de temperaturas es necesario declarar un contenedor (un ArayList en este caso) que será el encargado de llevar a cabo esta tarea a través de los métodos registrarObservador y eliminarObservador.

Por otro lado, hemos definido dos clases diferentes, DispositivoMovil y WidgetOrdenador, que implementan la interfaz Observador descrita en el patrón. Aunque por sencillez ambas clases son prácticamente iguales, nótese como, en un caso real, éstas podrían estar encargadas de realizar funcionalidades completamente diferentes. En nuestro caso, la única diferencia va a estar en el mensaje que van a mostrar en pantalla cuando se produzca un cambio en el objeto observado.



Apartado c

- **Interfaz Sujeto**

```
public interface Sujeto {

	public void registrarObservador(Observador o);
	public void eliminarObservador(Observador o);
	public void notificarObservadores();
}
```

- **Interfaz Observador**

```
public interface Observador {
	public void actualizar(float temperatura);
}
```

- **Clase EstaciónTemperaturas**

```
public class EstacionTemperaturas implements Sujeto {

	private ArrayList<Observador> observadores;
	private float temperatura;

	public EstacionTemperaturas() {
		observadores = new ArrayList<Observador>();
	}

	public void registrarObservador(Observador o) {
		observadores.add(o);
	}
	public void eliminarObservador(Observador o) {
		int i = observadores.indexOf(o);
		if (i >= 0) {
			observadores.remove(i);
		}
	}
	public void notificarObservadores() {

		for (int i = 0; i < observadores.size(); i++) {
			Observador o = observadores.get(i);
			o.actualizar(temperatura);
		}
	}
	public float obtenerTemperatura() {
		return temperatura;
	}
	public void actualizarTemperatura(float temperatura) {
		this.temperatura = temperatura;
		this.notificarObservadores();
	}
}
```

- **Clase DispositivoMóvil**

```
public class DispositivoMovil implements Observador {

	private Sujeto estacionTemperaturas;
	private float temperatura;

	public DispositivoMovil(Sujeto estacionTemperaturas) {
		this.estacionTemperaturas = estacionTemperaturas;
		this.estacionTemperaturas.registrarObservador(this);

	}
	public void actualizar(float temperatura) {

		this.temperatura = temperatura;
		mostrar();
	}
	public void mostrar() {
		System.out.println("La temperatura en el dispositivo móvil ha cambiado a " + temperatura
				+ " Grados");
	}
}
```

- **Clase WidgetOrdenador**

```
public class WidgetOrdenador implements Observador {

	private Sujeto estacionTemperaturas;
	private float temperatura;

	public WidgetOrdenador(Sujeto estacionTemperaturas) {
		this.estacionTemperaturas = estacionTemperaturas;
		this.estacionTemperaturas.registrarObservador(this);

	}

	public void actualizar(float temperatura) {

		this.temperatura = temperatura;
		mostrar();
	}

	public void mostrar() {
		System.out.println("La temperatura en el Widget del ordenador ha cambiado a " +
temperatura
				+ " Grados");
	}
}
```

- **Clase Test para probar todo el código**

```
public class Test {


	public static void main(String[] args) {
		EstacionTemperaturas estacion=new EstacionTemperaturas();

		WidgetOrdenador wo=new WidgetOrdenador(estacion);
		DispositivoMovil dm=new DispositivoMovil(estacion);

		estacion.actualizarTemperatura(20);
		estacion.actualizarTemperatura(30);
		estacion.actualizarTemperatura(40);


	}

}
```