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)

SEPTIEMBRE 2013 – MODELO A – 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: Usamos el patrón Builder (Constructor) cuando:

a) Queremos permitir diferentes representaciones del objeto que está siendo construido.
b) No queremos permitir diferentes representaciones del objeto que está siendo construido.
c) Queremos usar un algoritmo de construcción que no sea independiente de las partes del objeto que está siendo construido.
d) No queremos aislar el código de construcción y representación.

Pregunta 2: El patrón Singleton (Único) garantiza que:

a) Las instancias de una clase están accesibles a un solo cliente.
b) Una clase sólo tenga una instancia y proporciona un punto de acceso global a ella.
c) No se puede generar subclases del patrón
d) Una clase tenga más que una instancia.

Pregunta 3: Los beneficios del patrón Composite (Compuesto) son (indica la respuesta falsa):

a) Poder definir jerarquías de clases primitivos y compuestos.
b) Facilita añadir nuevos tipos de componentes.
c) Simplifica el diseño del cliente.
d) Hacer más fácil restringir los componentes de un compuesto.

Pregunta 4: Las situaciones comunes para utilizar el patrón Proxy (Apoderado) son (indica la respuesta falsa):

a) Un proxy local que proporciona un representante local de un objeto local.
b) Un proxy virtual que crea objetos costosos por encargo.
c) Un proxy remoto que proporciona un representante local de un objeto remoto.
d) Un proxy de protección que controla el acceso a un objeto.

Pregunta 5: Las participantes del patrón Interpreter (Intérprete) incluyen (indica la respuesta falsa):

a) ExpresionTerminal
b) ExpresionAbstracta
c) Contexto
d) Interprete

Pregunta 6: Las participantes del patrón Observer (Observador) incluyen (indica la respuesta falsa):

a) Observador
b) ObservadorConcreto
c) Sujeto
d) SujetoConcreto

Pregunta 7: Dado este fragmento de código:

```
public class Test {
    String n;
    Test(String i) {
        n = i;
    }
    Test() {
        this("Prueba1");
    }
    public static void main(String args[]) {
        Test p1 = new Test("Prueba2");
```

```
        Test p2 = new Test();
    }
}
```

Después de ejecutar el método main, ¿qué valor tendría la variable n para los objetos p1 y p2?

a. Prueba2, Prueba1
b. Prueba2, Prueba2
c. Prueba1, Prueba1
d. Prueba2, null

Pregunta 8: Dado este fragmento de código:

```
import java.util.*;
public class Foo {
   String[] letras = {"a","b","c"};
   List<String> letrasList;

   Foo() {
      letrasList = ???
   }
}
```

¿Cómo se puede convertir el arreglo (array) letras en la lista letrasList?

a. letrasList = new List<String>(letras);
b. letrasList = new Arrays(letras);
c. letrasList = Arrays.asList(letras);
d. letrasList = Arrays.newList(letras);

Pregunta 9: ¿Cuál es la diferencia entre el acoplamiento y la cohesión?

a. El acoplamiento describe cuánto se ajusta una unidad de código a una tarea y la cohesión describe interconectividad de las clases.
b. El acoplamiento describe la interconectividad de las clases y la cohesión describe cuánto se ajusta una unidad de código a una tarea.
c. El acoplamiento describe la interconectividad de las clases y la cohesión describe el grado de visibilidad de las variables de las clases.
d. El acoplamiento y la cohesión hacen referencia a dos aspectos complementarios de la interconectividad de las clases.

Pregunta 10: ¿Cómo se hace la depuración en BlueJ?

a. A través del menú de herramientas.
b. Haciendo clic con el botón derecho sobre un objeto y seleccionando la entrada Debugger.
c. A través del menú Project.
d. Incorporando un punto de ruptura en el código fuente y ejecutando el código correspondiente.

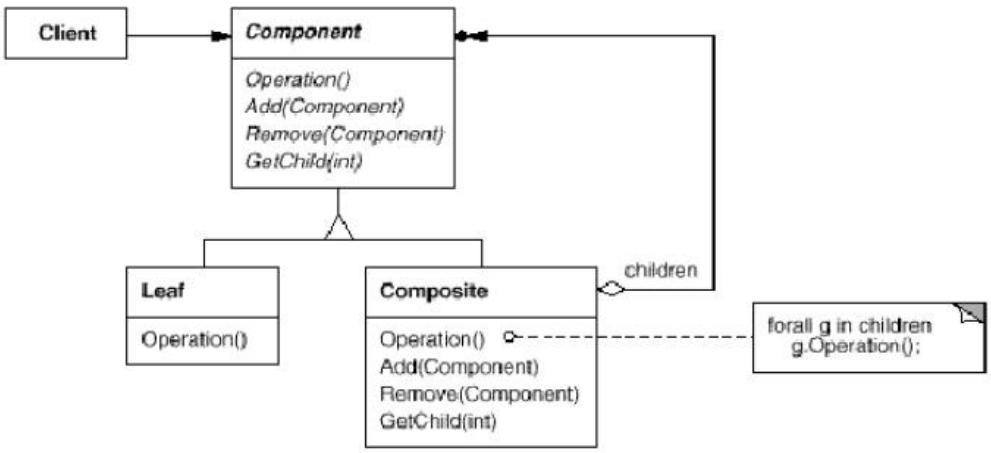
PARTE PRÁCTICA [6,5 PUNTOS]:

Pregunta 11: Se podría usar el patrón Compound (Compuesto) para un programa que saca información en la pantalla del ordenador y en la impresora a la vez:

a) **[1 punto]** Justifique la elección de este patrón y analiza las ventajas y desventajas de la elección. ¿Hay otro patrón que se puede haber usado en vez 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, así como las clases reutilizadas de su práctica original (en el caso de haber reutilizado alguna).
c) **[1,25 puntos]** Implemente en Java el diagrama de clases propuesto.

a) El patrón Compound (Compuesto, pág 151 del libro de la asignatura) permite construir una estructura de objetos en forma de un árbol que son similares y que se pueden tratar de la misma forma. En este ejemplo nos permite usar un método pintar para presentar la información al usuario independientemente de si se trata de la pantalla del ordenador o de la impresora. Y la ventaja de usar el patrón es que el código en el cliente es muy sencillo. La desventaja del patrón es que una vez que se ha definido la estructura del árbol su comportamiento es muy general y no es posible restringirlo a un subconjunto de objetos. Otros patrones relacionados que podrían aportar algo de la funcionalidad de Compound son el Flyweight y Visitor.

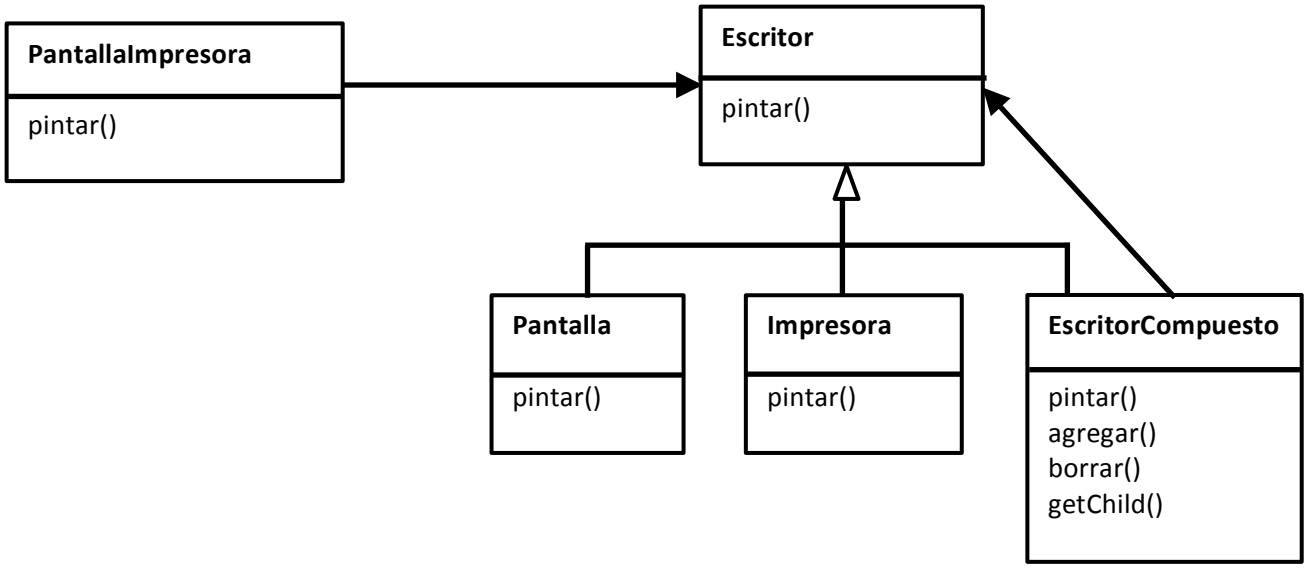
b) El patrón Compound tiene la siguiente estructura:



Y los participantes son:

- **Component:** Declara la interfaz para los objetos de la composicion, es la interfaz de acceso y manipulacion de los componentes hijo e implementa algunos comportamientos por defecto.
- **Client:** Manipula objetos atravez de la interfaz proporcionada por Component.
- **Composite:** Define el comportamiento de los componentes compuestos, almacena a los hijos e implementa las operaciones de manejo de los componentes.
- **Leaf:** Definen comportamientos para objetos primitivos del compuesto.

Así que en este caso la estructura del patrón sería:



c)

```
import java.util.*;

abstract class Escritor {
	protected String nombre;

	public Escritor(String nombre) {
		this.nombre = nombre;
	}

	public String getNombre() {
		return(nombre);
	}

	abstract public void pintar(String s);
}

class EscritorCompuesto extends Escritor {
	private ArrayList<Escritor> escritores = new ArrayList<Escritor>();

	public EscritorCompuesto(String name) {
		super(name);
	}

	public void agregar(Escritor componente) {
		escritores.add(componente);
	}

	public void eliminar(Escritor componente) {
		escritores.remove(componente);
	}

	public Escritor getChild(String n) {
		for (Escritor e: escritores) {
			if (e.getNombre().equals(n))
				return(e);
		}
		return(null);
	}
```

```
	public void pintar(String s) {
		for (Escritor e: escritores) {
			e.pintar(s);
		}
	}
}

class Pantalla extends Escritor {

	public Pantalla() {
		super("Pantalla");
	}

	public void mostrar(int depth) {
		System.out.println('-' + "" + nombre);
	}

	public void pintar(String s) {
		System.out.println("En Pantalla, pintando: " + s);
	}
}

class Impresora extends Escritor {

	public Impresora() {
		super("Impresora");
	}

	public void mostrar(int depth) {
		System.out.println('-' + "" + nombre);
	}

	public void pintar(String s) {
		System.out.println("En Impresora, pintando: " + s);
	}
}

public class PantallaImpresora {
	public static void main(String[] args) {
		EscritorCompuesto ordenador = new EscritorCompuesto("perifericos");
		ordenador.agregar(new Pantalla());
		ordenador.agregar(new Impresora());
		ordenador.pintar("Hola Mundo");
	}
}
```

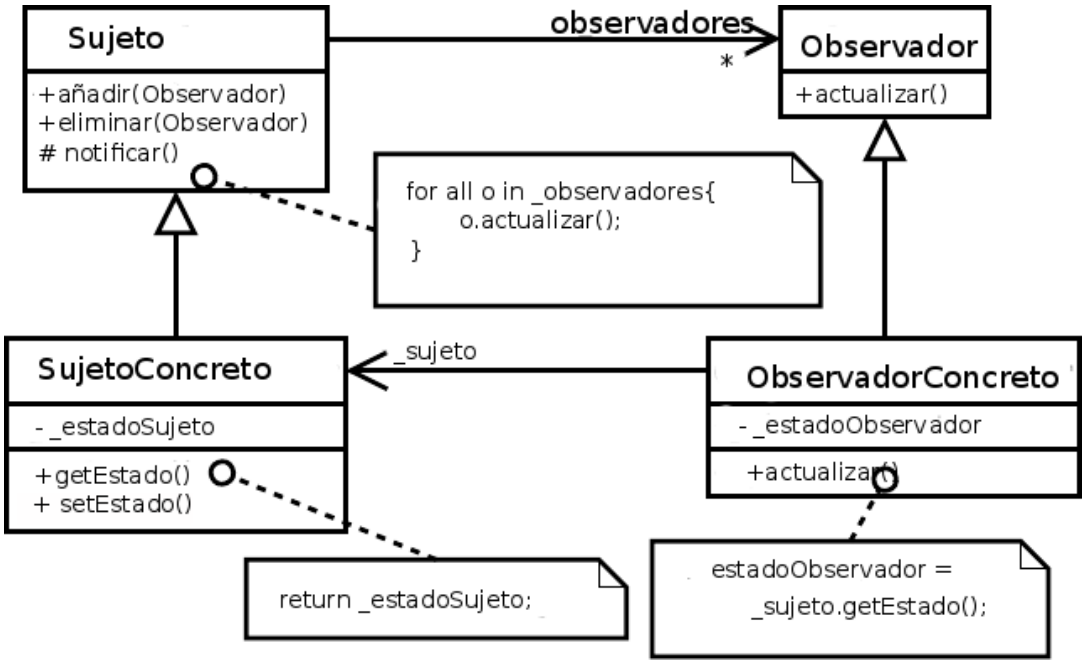
Pregunta 12: Se podría usar el patrón Observer (Observador) para un sistema de gestión de una Biblioteca, donde cada vez que un lector devuelve un libro hay que avisar a otras partes del sistema que están interesadas en conocer este evento:

a) **[1 punto]** Justifique la elección de este patrón y analiza las ventajas y desventajas de la elección. ¿Hay otro patrón que se puede haber usado en vez 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, así como las clases reutilizadas de su práctica original (en el caso de haber reutilizado alguna).
c) **[1,25 puntos]** Implemente en Java el diagrama de clases propuesto.

a) El patrón Observer (Observador, pág 269 del libro de la asignatura) conocido también como el patrón publicación-inscripción define una dependencia del tipo uno-a-muchos. Así que cuando un objeto cambia su estado, notifica este cambio a los dependientes. En este caso es el patrón ideal porque su comportamiento refleja el escenario donde la devolución de un libro a la biblioteca es la condición para informar a todas las partes del sistema que tiene que actualizar su estado (como por ejemplo, el catálogo central de libros, sistema de avisos a lectores que están esperando al libro, etc.). La ventaja de este patrón incluye que permite la modificación del sujeto y los observadores de una forma independiente y que se puede añadir nuevos observadores sin la necesidad de modificar el sujeto. La desventaja incluye el acoplamiento abstracto que hay entre el sujeto y sus observadores y la dificultad en saber el efecto de actualizaciones del sistema. Otros patrones relacionados que podrían aportar algo de la funcionalidad de Compound son el Mediator y el Singleton.

b)

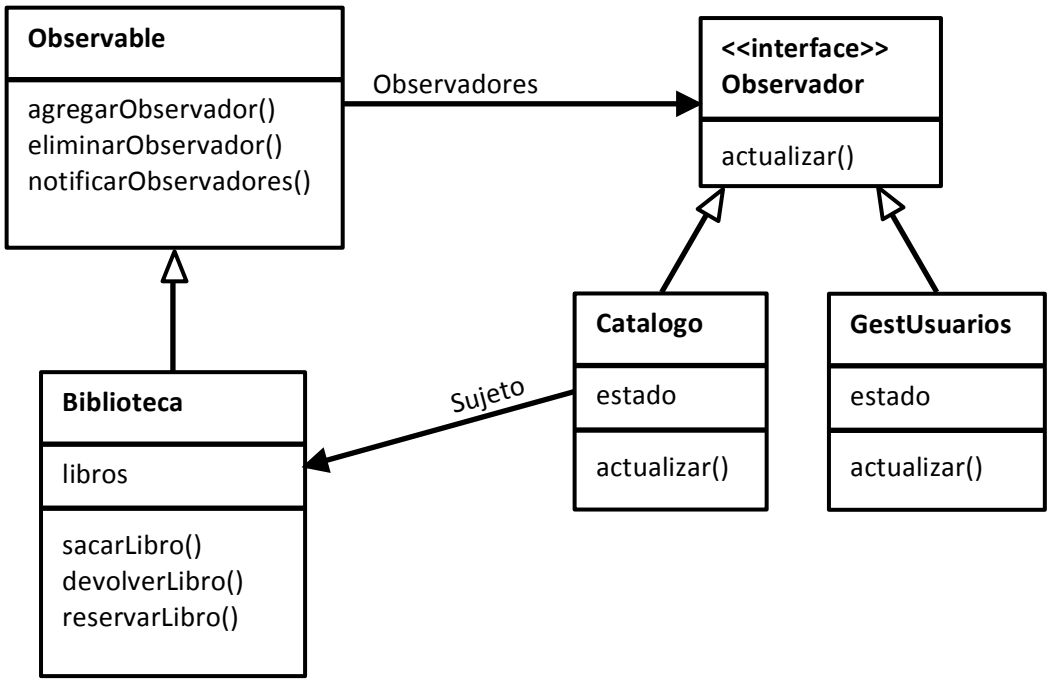
El patrón Observer tiene la siguiente estructura:



Los participantes son:

- **Sujeto (Subject):** El sujeto concreto proporciona una interfaz para agregar (attach) y eliminar (detach) observadores. El Sujeto conoce a todos sus observadores.
- **Observador (Observer):** Define el método que usa el sujeto para notificar cambios en su estado (update/notify).
- **Sujeto Concreto (ConcreteSubject):** Mantiene el estado de interés para los observadores concretos y los notifica cuando cambia su estado. No tienen porque ser elementos de la misma jerarquía.
- **Observador Concreto (ConcreteObserver):** Mantiene una referencia al sujeto concreto e implementa la interfaz de actualización, es decir, guardan la referencia del objeto que observan, así en caso de ser notificados de algún cambio, pueden preguntar sobre este cambio.

Así que en este caso la estructura del patrón sería:



c)

```
import java.util.ArrayList;

abstract class Observable {
	private ArrayList<Observador> observadores;
	public Observable() {
		observadores = new ArrayList<Observador>();
	}

	public void agregarObservador(Observador o) {
		observadores.add(o);
	}

	public void eliminarObservador(Observador o) {
		observadores.remove(o);
	}

	public void notificarObservadores(Libro l) {
		for (Observador o : observadores) {
			o.actualizar(l);
		}
	}
}

class Libro {
	private String titulo;
	private String referencia;

	public Libro(String tit, String ref) {
		titulo = tit;
		referencia = ref;
	}

	public String getTitulo() {
```

```
		return titulo;
	}
	public void setTitulo(String titulo) {
		this.titulo = titulo;
	}
	public String getReferencia() {
		return referencia;
	}
	public void setReferencia(String referencia) {
		this.referencia = referencia;
	}
}

class MiBiblioteca extends Observable {

	public MiBiblioteca(){
		super();
	}

	public void devolverLibro(Libro l){
		notificarObservadores(l);
	}
}

public class Biblioteca {

	public static void main(String[] args) {
		MiBiblioteca b = new MiBiblioteca();
		b.agregarObservador(new Catalogo());
		b.agregarObservador(new GestUsuarios());
		b.devolverLibro(new Libro("Patrones de Diseño","foo-12345"));
	}
}

interface Observador {
	public void actualizar(Libro l);
}

class Catalogo implements Observador {
	public void actualizar(Libro l){
		System.out.println("En Catálogo...");
		System.out.println("Libro: " + l.getTitulo() + " (" + l.getReferencia() + ") ha sido devuelto");
	}
}

class GestUsuarios implements Observador {
	public void actualizar(Libro l){
		System.out.println("En GestUsuarios...");
		System.out.println("Libro: " + l.getTitulo() + " (" + l.getReferencia() + ") es disponible para usurio");
	}
}
```