UNIVERSIDAD DE MÁLAGA

Modelado y Diseño de Software

Ejemplo de Examen 2017-18 (5)

Nombre: _____

PARTE 1 (15 minutos)

A. Define los siguientes conceptos

- "Ingeniería de software"
- "Modelo"
- "Diseño de software"
- "Patrón de diseño"

B Indica los principales elementos de los diagramas de despliegue de UML. [0.25]

C. Define el concepto de "Antipattern" (anti-patrón de diseño), menciona los tres tipos que hay, y describe tres antipatterns. [0.25]

PARTE 2 (3 horas 45 minutos)

1. El metamodelo de Java

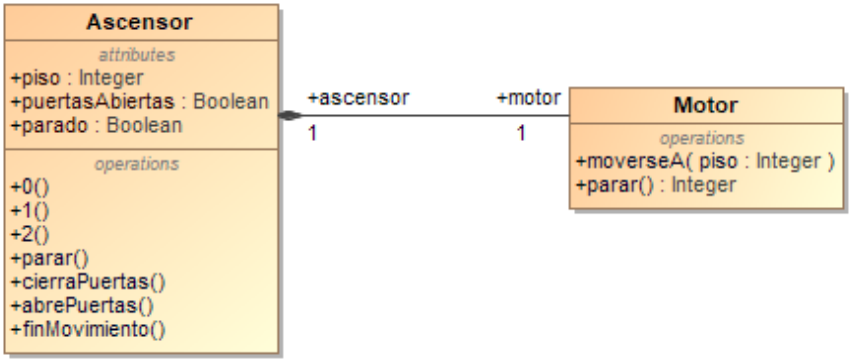
Utilizando USE, define un metamodelo con la sintaxis abstracta de un subconjunto mínimo del lenguaje Java, que contenga la definición de clases, sus atributos y métodos. En particular, ha de ser capaz de representar al menos los siguientes conceptos: Tipo Básico (Integer, Real, etc.), Clase, Atributo, Método y Parámetro (de un método). Los atributos, métodos y parámetros han de tener un nombre y un tipo, que puede ser un tipo básico o una clase previamente definida. Asimismo, ha de ser posible describir la cardinalidad y visibilidad de los elementos, sus nombres, y las relaciones de herencia entre clases.

2. El ascensor

Supongamos la siguiente clase, Ascensor, que representa un ascensor de un bloque de con 2 pisos y una planta baja (o piso 0). Sus atributos indican el piso en el que actualmente se encuentra ("piso"=0, 1, 2), si está "parado" (o moviéndose), y si tiene las puertas abiertas o cerradas. Cada ascensor tiene asociado un motor, al que le indica que debe mover la cabina y desplazarla a un piso. Una vez el motor acaba el movimiento solicitado, se lo indica al ascensor usando la operación "`finMovimiento()`". El ascensor también puede ordenar al motor que se pare cuando está moviéndose, usando la operación "`parar()`", que espera a que se detenga la cabina y devuelve el número del piso en el que el motor finalmente se detiene la cabina. La operación `parar()` no tiene efecto si el motor no está en marcha. El resto de operaciones son las que representan los botones de la cabina o las acciones que puede hacer el ascensor (por ejemplo, abrir o cerrar las puertas, que puede hacerlo tanto un usuario del ascensor como el propio ascensor).

a) Se pide especificar en UML, con MagicDraw, un diagrama de estados del `Ascensor` que describa sus posibles estados, suponiendo que los botones del ascensor no tienen memoria (es decir, si no pueden realizar una acción en un momento dado, el sistema no pospone su actuación sino que pierden su efecto). Dicho diagrama ha de describir su comportamiento al recibir cada uno de los mensajes en cada uno de los estados posibles, de acuerdo al siguiente comportamiento:

- Mientras se está moviendo el ascensor los botones (salvo el de parar) no tienen efecto.
- Para poder moverse a un piso usando los botones 0, 1 o 2, el ascensor tiene que estar con la puerta cerrada
- Al pulsar el botón de un piso distinto al que se encuentra el ascensor, cuando la puerta está cerrada, el ascensor se desplaza hasta ese piso. Cuando llega el propio ascensor se encarga de abrir las puertas.
- Si se pulsa el botón del piso en curso, se abren las puertas.



b) Especificar en UML un diagrama de secuencia que represente el intercambio de mensajes que ocurre cuando el ascensor está parado en el piso 1 y con las puertas abiertas, y una persona usa el ascensor para ir a la planta 2. Utilícese un actor "`Usuario`" para representar a la persona que usa dicho ascensor y pulsa los correspondientes botones.

c) Queremos diseñar ese sistema para su posterior implementación en Java, utilizando el patrón de diseño Estado. Se pide:

- Discutir la mejor forma de implementar los atributos de las clases de forma que no sean públicos, y la relación entre el ascensor y el motor.
- Realizar un diagrama de clases UML con el esquema básico del diseño propuesto.
- Implementar (de forma esquemática) dicha solución en Java.

3. Sistema de ventanas

Supongamos que estamos desarrollando un sistema de ventanas, y que una de las clases que forma parte del sistema es la clase `Texto`, que sirve para representar un texto en una determinada área rectangular de la pantalla, para lo cual la clase dispone de un método `dibujar()`.

Según vamos desarrollando nuestro sistema, incorporamos diversos adornos a nuestras ventanas, para lo que especializamos la clase `Texto` en `TextoColoreado` (en el que podemos determinar el color del fondo), `TextoEnmarcado` (en el que representamos el texto con un borde alrededor para resaltarlo), `TextoConScroll` (que permite hacer scrolling en la ventana cuando el texto es demasiado largo), etc. En todas estas clases redefinimos el método `dibujar()` para adaptarlo a las nuevas características de la clase.

El problema surge cuando nuestros clientes nos sugieren la posibilidad de tener textos coloreados con scroll; textos coloreados y enmarcados; textos coloreados, enmarcados y con scroll. etc. La estrategia de seguir especializando la clase Texto para cada una de las posibles combinaciones de adornos deja de parecer la solución adecuada.

Proponer una solución alternativa a este problema en la que, a pesar de que vayamos añadiendo nuevos adornos a nuestras ventanas, no se produzca una explosión combinatoria del número de clases en el sistema.

(a) Discutir las ventajas e inconvenientes de la solución propuesta,
(b) Realizar un diagrama de clases UML con el esquema básico del diseño propuesto
(c) Implementar (de forma esquemática) dicha solución en Java.

Puntuaciones: 1 (1.5) 2 (2+1+2) 3 (0.5+1.5+1). Examen reducido: 1 y 2.

Soluciones al Examen 5

Parte 1. A.

- La *ingeniería de software* es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software.
- *Modelo*: Una representación o especificación de un sistema, desde un determinado punto de vista y con un objetivo concreto.
- *Diseño*: Conjunto de planes y decisiones para definir un producto con los suficientes detalles como para permitir su realización física de acuerdo a unos requisitos
- *Patrón de Diseño*: Una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de software.

Parte 1.B. Indica los principales elementos de los diagramas de despliegue de UML.

Un Diagrama de Despliegue modela la arquitectura y topología de comunicación de un sistema distribuido en tiempo de ejecución. Muestra la disposición física de los artefactos software en nodos, las conexiones entre los nodos, y la asignación de artefactos y componentes software a cada nodo. Los principales elementos son:

- Nodo: Un nodo modelan un recurso computacional del sistema (hardware o software) en tiempo de ejecución. Se representan mediante un hexaedro (cubo) y pueden contener a otros nodos, a artefactos y a componentes software.
- Asociación: En el contexto del diagrama de despliegue, una asociación representa una ruta de comunicación entre nodos.
- Artefacto: Un artefacto es un producto del proceso de desarrollo de software, que puede incluir archivos fuente, ejecutables, documentos de diseño, reportes de prueba, prototipos, manuales de usuario e incluso los propios los modelos del sistema y/o del proceso.
- También son elementos de los diagramas de despliegue las instancias de los nodos, de las asociaciones y de los artefactos.

Además de estos elementos, en los diagramas de despliegue pueden incluirse otros elementos como por ejemplo instancias de componentes software, representando su despliegue en los nodos del sistema.

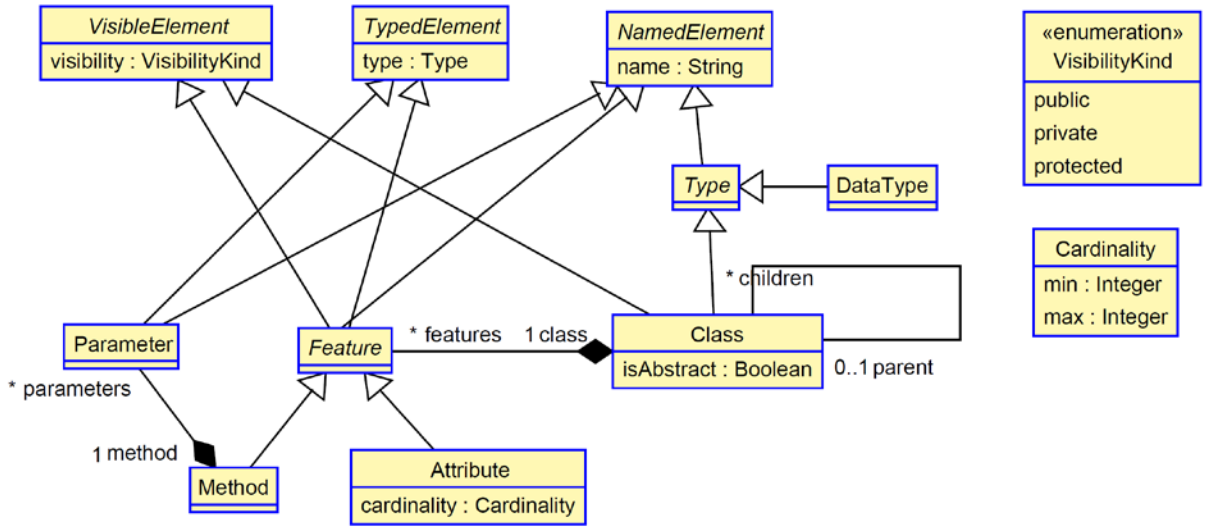
Parte 1.C. Define el concepto de "Antipattern" (anti-patrón de diseño), menciona los tres tipos que hay, y describe tres antipatterns.

Los AntiPatterns identifican una serie de procesos y estructuras defectuosos que son muy comunes en la arquitectura, implementación o gestión de proyectos software y que son la causa de problemas y errores conocidos y recurrentes. Hay tres tipos de antipatterns:

- Antipatterns de desarrollo: que aparecen en los diseños de software y en los programas; por ejemplo el Blob.
- Antipatterns de arquitectura: que aparecen en la arquitectura software de los sistemas; por ejemplo el Vendor Lock-in.
- Antipatterns de gestión: aparecen en la gestión de proyectos; por ejemplo el Analysis paralysis.

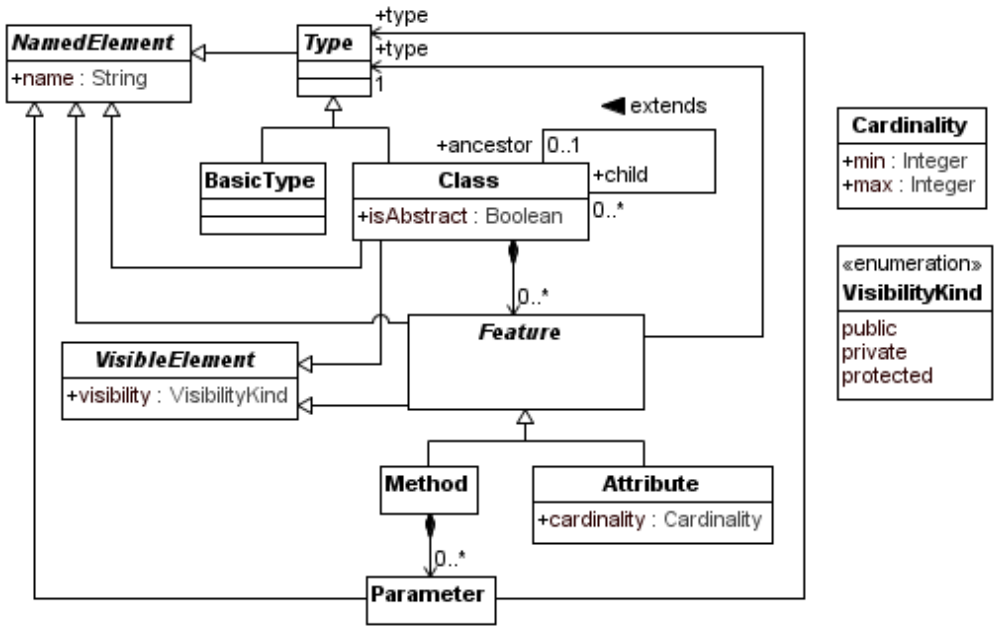
Ejercicio 1

Solución:




```
model Java

enum VisibilityKind {public, private,
protected}

class Cardinality
attributes
  min:Integer
  max:Integer
end

abstract class NamedElement
attributes
  name : String
end

abstract class VisibleElement
attributes
  visibility:VisibilityKind
end

abstract class Type < NamedElement
end

abstract class TypedElement
attributes
  type:Type
end

class DataType < Type
end

class Class < Type, VisibleElement
attributes
  isAbstract:Boolean
end

abstract class Feature < NamedElement,
                VisibleElement, TypedElement
end

class Method < Feature
end

class Attribute < Feature
attributes
  cardinality:Cardinality
end

class Parameter < NamedElement, TypedElement
end

composition Features between
   Class [1] role class
   Feature [*] role features
end

composition Methods between
   Method [1] role method
   Parameter [*] role parameters
end

association Extension between
   Class [0..1] role parent
   Class [*] role children
end

constraints
context Class inv noCycles:
  self.parent->closure->excludes(self)
```

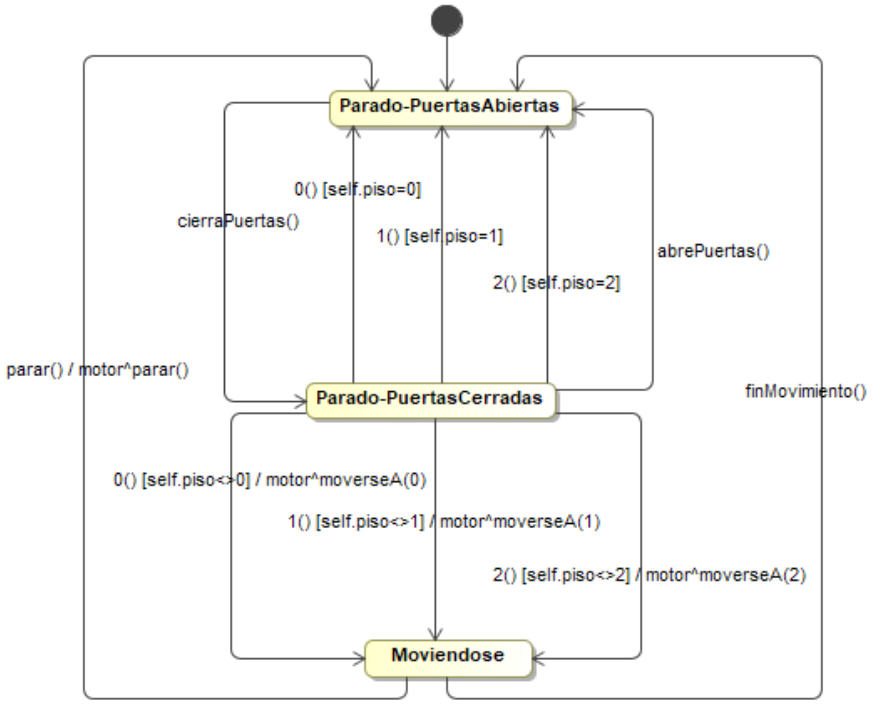
Por supuesto, existen otras soluciones como por ejemplo la siguiente (expresada en MagicDraw):



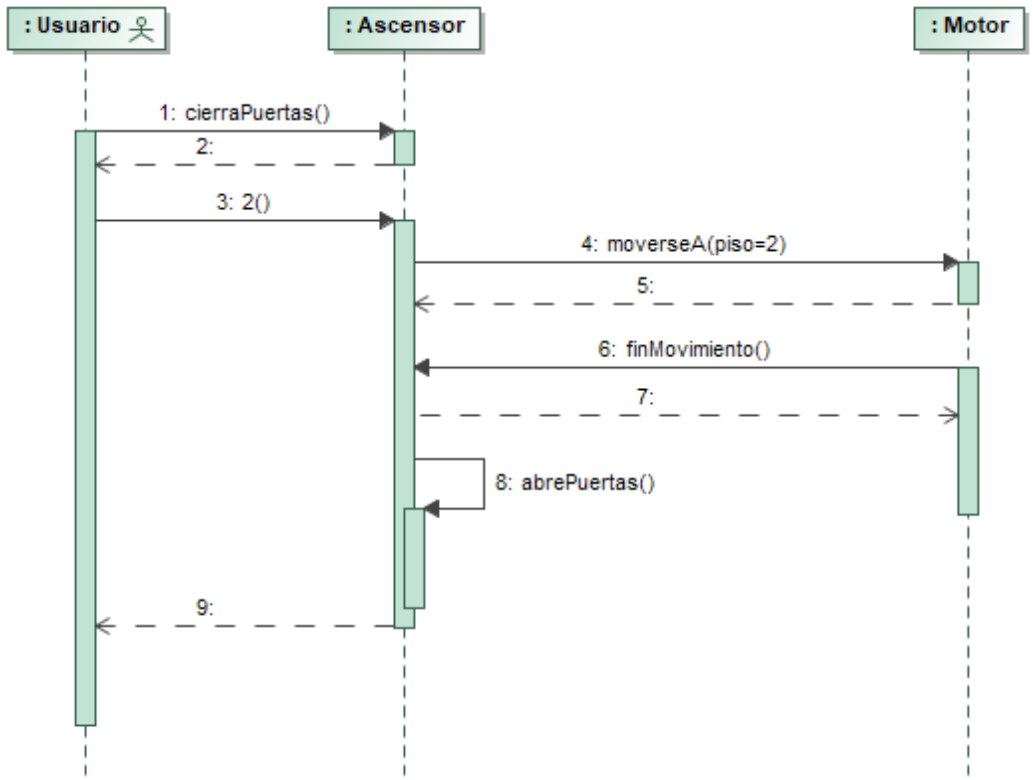
Ejercicio 2

a) El siguiente diagrama de estados muestra una posible solución. Por legibilidad, en el diagrama no se muestran las transiciones para aquellas operaciones que no tienen efecto alguno, como por ejemplo la invocación de las operaciones `0()`, `1()`, `2()`, `parar()`, `abrePuertas()` y `finMovimiento()` en el estado "`Parado-PuertasAbiertas`" (la única operación admisible en ese estado es "cierrapuertas()"

Obsérvese que esta no es la única solución, pues también podrían haberse utilizado sólo 2 estados principales: Parado y Moviendose, uno de los cuales (Parado) sería un estado compuesto con dos subestados: PuertasAbiertas y PuertasCerradas. Sin embargo, en esta solución se ha optado por no usar estados compuestos.



b) Solución:

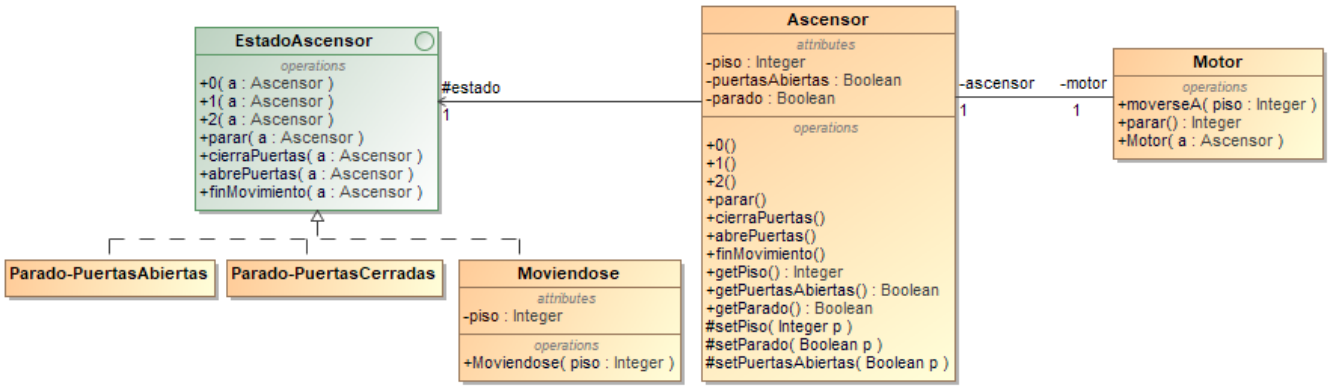


c) Solución:

Para que los atributos de las clases no sean públicos lo mejor es establecer getters and setters, y definir la visibilidad de dichas operaciones. Por ejemplo, los getters pueden ser públicos pero los setters no deben serlo, pues eso permitiría a cualquier otro objeto cambiar el valor del atributo "piso" o de "parado", lo que dejaría al objeto en un estado inconsistente. Una solución consistiría en hacer protected los getters.

En cuanto a las relaciones entre el ascensor y su estado y motor, pueden usarse relaciones entre ellos, que luego pueden convertirse en atributos en java.

El siguiente diagrama muestra un posible diseño



Finalmente, el esquema de una posible implementación en Java que hace uso de ese diseño podría ser la siguiente (obsérvese que los nombres de las operaciones `0()`, `1()` y `2()` se han sustituido por `cero()`, `uno()` y `dos()` ya que Java no permite este tipo de nombres):

```java
//Interface con el Estado
public interface EstadoAscensor {
	public void cero(Ascensor a);
	public void uno(Ascensor a);
	public void dos(Ascensor a);
	public void parar(Ascensor a);
	public void abrePuertas(Ascensor a);
	public void cierraPuertas(Ascensor a);
	public void finMovimiento(Ascensor a);
}
```

```java
//Clase Ascensor
public class Ascensor {
	protected EstadoAscensor estado;
	private int piso;
	private boolean puertasAbiertas;
	private boolean parado;
	private Motor motor;

	public Ascensor(){
		piso = 0;
		parado = true;
		puertasAbiertas = true;
		estado = new ParadoPuertasAbiertas();
		motor = new Motor (this);
	}

	//getters and setters
	public int getPiso() {
		return piso;
	}
	public boolean getPuertasAbiertas(){
		return puertasAbiertas;
	}
	public boolean getParado(){
		return parado;
	}
	public Motor getMotor(){
		return motor;
	}
	protected void setPiso(int p) {
		piso = p;
	}
	protected void setPuertasAbiertas(boolean p){
		puertasAbiertas = p;
	}
	protected void setParado(boolean p){
		parado = p;
	}
	//operations -- delegated to the state
	public void cero(){
		estado.cero(this);
	}
	public void uno(){
		estado.uno(this);
	}
	public void dos(){
		estado.dos(this);
	}
	public void parar(){
		estado.parar(this);
	}
	public void abrePuertas(){
		estado.abrePuertas(this);
	}
	public void cierraPuertas(){
		estado.cierraPuertas(this);
```

```
	}
	public void finMovimiento(){
		estado.finMovimiento(this);
	}
}
```

```
public class Motor {
	private Ascensor ascensor;
	public Motor(Ascensor a) {
		ascensor = a;
	}
	public void moverseA(int piso){
	//TODO:
	}
	public int parar(){
	//TODO:
		return /* piso en donde para finalmente */ 1;
	}
}
```

```
public class ParadoPuertasAbiertas implements EstadoAscensor {

	@Override
	public void cero(Ascensor a){
		//no action
	}
	@Override
	public void uno(Ascensor a){
		//no action
	}
	@Override
	public void dos(Ascensor a){
		//no action
	}
	@Override
	public void parar(Ascensor a){
		//no action
	}
	@Override
	public void abrePuertas(Ascensor a){
		//no action
	}
	@Override
	public void cierraPuertas(Ascensor a){
		a.estado = new ParadoPuertasCerradas();
		a.setPuertasAbiertas(false);
	}
	@Override
	public void finMovimiento(Ascensor a){
		//no action
	}
}
```

```
public class ParadoPuertasCerradas implements EstadoAscensor {
	@Override
	public void cero(Ascensor a){
		if (a.getPiso()!=0) {
			Motor m = a.getMotor();
			m.moverseA(0);
			a.estado=new Moviendose(0);
			a.setParado(false);
			a.setPuertasAbiertas(false);
		} //else no action needed
	}
	@Override
```

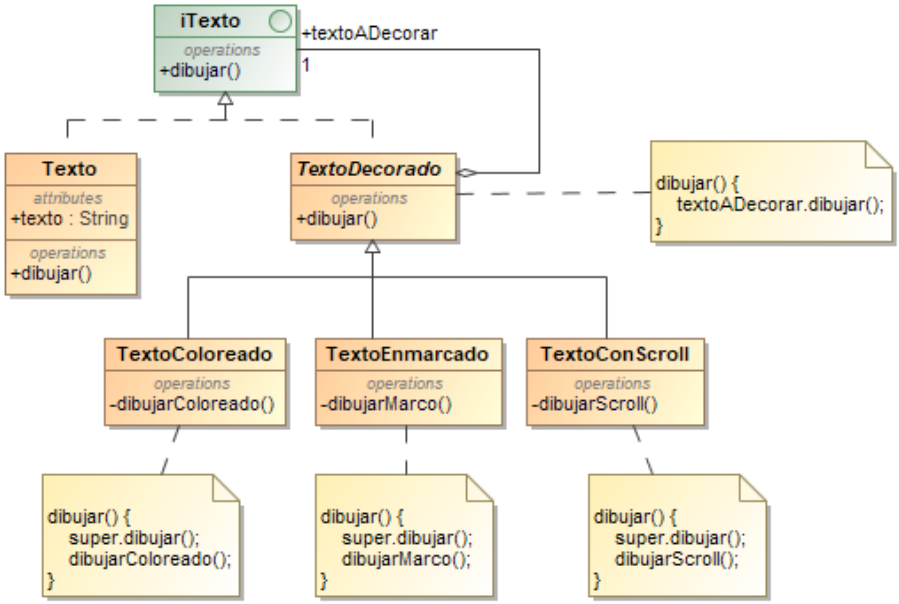
```
	public void uno(Ascensor a){
		if (a.getPiso()!=1) {
			Motor m = a.getMotor();
			m.moverseA(1);
			a.estado=new Moviendose(1);
			a.setParado(false);
			a.setPuertasAbiertas(false);
		} //else no action needed
	}
	@Override
	public void dos(Ascensor a){
		if (a.getPiso()!=2) {
			Motor m = a.getMotor();
			m.moverseA(2);
			a.estado=new Moviendose(2);
			a.setParado(false);
			a.setPuertasAbiertas(false);
		} //else no action needed
	}
	@Override
	public void parar(Ascensor a){
		//no action
	}
	@Override
	public void abrePuertas(Ascensor a){
		a.estado=new ParadoPuertasAbiertas();
		a.setPuertasAbiertas(true);
	}
	@Override
	public void cierraPuertas(Ascensor a){
		//no action
	}
	@Override
	public void finMovimiento(Ascensor a){
		//no action
	}
}
```

```
public class Moviendose implements EstadoAscensor {

	private int piso = 0; // piso al que se ha de mover

	public Moviendose(int p){
		piso = p;
	}
	@Override
	public void cero(Ascensor a) {
		//no action
	}
	@Override
	public void uno(Ascensor a) {
		//no action
	}
	@Override
	public void dos(Ascensor a) {
		//no action
	}
	@Override
	public void parar(Ascensor a) {
		Motor m = a.getMotor();
		int p = m.parar(); // this operation waits for the motor to stop
		a.estado=new ParadoPuertasAbiertas();
		a.setPiso(p);
		a.setPuertasAbiertas(true);
		a.setParado(true);
	}
```

```
        @Override
        public void abrePuertas(Ascensor a) {
                //no action
        }
        @Override
        public void cierraPuertas(Ascensor a) {
                //no action
        }
        @Override
        public void finMovimiento(Ascensor a) {
                a.setParado(true);
                a.setPuertasAbiertas(true);
                a.estado=new ParadoPuertasAbiertas();
                a.setPiso(piso);
        }
}
```

Ejercicio 3

Para solucionar este problema aplicamos el patrón de diseño "Decorador" que precisamente sirve para aplicar diversas funcionalidades de forma ordenada y composicional. El esquema sería el siguiente.



a) Este esquema presenta ventajas interesantes frente a la solución sin patrones, en donde tendríamos una explosión combinatoria de clases cada vez que se quisiera implementar un sistema concreto. Este patrón permite añadir responsabilidades a un objeto de forma dinámica y proporciona una alternativa flexible a la subclasificación para añadir funcionalidad.

b) El diagrama de clases es el que se ha indicado anteriormente.

c) En Java es posible implementar esta solución como sigue:

```
public interface iTexto {
    void dibujar(); // Dibuja el texto
}

// Implementación de un texto simple sin funcionalidad alguna
class Texto implements iTexto {
    String texto;
    Public Texto (String texto) {
        This.texto = texto;
    }
```

```
    public void dibujar() {
        // Aquí se dibujaría el texto como se quiera, en su forma más simple.
       // Por ejemplo: system.out.println(texto);
    }
}

// Las siguientes clases contienen los decoradores para todos los tipos de textos,
// incluyendo a los propios decoradores

// La primera es la clase abstracta de la heredan todos.
// Nótese que implementa la interfaz iTexto.
abstract class TextoDecorado implements iTexto {
    protected iTexto textoADecorar; // el texto que hay que decorar

    public TextoDecorado (iTexto textoADecorar) {
        this. textoADecorar = textoADecorar;
    }

    public void dibujar() {
        textoADecorar.dibujar(); //Delegación
    }
}

// El primer decorador sería como sigue:
class TextoColoreado extends TextoDecorado {
    public TextoColoreado (iTexto textoADecorar) {
        super(textoADecorar);
    }
    @Override
    public void dibujar() {
        super.dibujar();
        dibujarColoreado ();
    }
   private void dibujarColoreado() {
        // Aquí vendría el código que dibuja con color
    }
}

// El segundo decorador sabe poner un marco
class TextoEnmarcado extends TextoDecorado {
    public TextoEnmarcado (iTexto textoADecorar) {
        super(textoADecorar);
    }
    @Override
    public void dibujar() {
        super.dibujar();
        dibujarEnmarcado();
    }
    private void dibujarEnmarcado() {
        // Aquí vendría el código que dibuja el marco
    }
}

// El tercer decorador sabe incorporar un scroll
class TextoConScroll extends TextoDecorado {
    public TextoConScroll (iTexto textoADecorar) {
        super(textoADecorar);
    }
    @Override
    public void dibujar() {
        super.dibujar();
        dibujarConScroll ();
    }

    private void dibujarConScroll () {
        // Aquí vendría el código que dibuja con posibilidades de scroll
    }
}
```

Finalmente, un ejemplo de programa que crea dibuja un texto decorado con todas las posibilidades es el siguiente:

```
public class DecoratedWindowTest {
  public static void main(String[] args) {
      // Crea un texto decorado con color, marco y scroll, en ese orden.
      iTexto textoDecorado =
        new TextoConScroll(new TextoEnMarcado(new TextoColoreado(new Texto(“Hola Mundo”))));
    }
}
```