Capítulo 12. Desarrollo de aplicaciones y patrones

A. J. Pérez

Desarrollo de aplicaciones y patrones

Características de una aplicación

Las características deseables para una aplicación de software son:

- **Mantenibilidad**
  Se dice que una *aplicación es mantenible* si está desarrollada de tal forma que el diseño de su código permite adaptarla a pequeñas (o no tan pequeñas) variantes que puedan surgir conforme a la *evolución del negocio*. Por ejemplo, la forma de calcular un impuesto, la manera de liquidar una comisión, el criterio para aceptar o rechazar la inscripción de un alumno a un curso, etcétera. *Negocio* se refiere al contexto en el que funciona la aplicación.

- **Escalabilidad**
  La *escalabilidad* hace referencia a cómo se incrementa el tiempo de respuesta o el consumo de recursos de una aplicación conforme aumenta la carga de trabajo.

  Se dice que una *aplicación es escalable* si tiene un diseño que permite, con un coste razonable, incrementar su rendimiento en volumen y capacidad de procesamiento utilizando otras tecnologías. Por ejemplo, si se tiene una aplicación que utiliza ficheros como almacén de datos y se adapta para que pueda funcionar con bases de datos. O bien, la aplicación trabaja con la base de datos MySQL y se quiere que funcione también con ORACLE, DB2 o DB4O.

- **Extensibilidad**
  Se dice que una *aplicación es extensible* si, con un coste razonable, se puede ampliar su funcionalidad a nuevos casos de uso que puedan surgir. Un *caso de uso* se refiere a las diferentes situaciones que contempla la aplicación.

Otras características de una aplicación son la *disponibilidad*, la *fiabilidad* y la *seguridad*, pero su estudio sobrepasa el alcance de este manual.

Cohesión y acoplamiento

Los términos de *cohesión* y *acoplamiento* son inseparables de la programación; complementan y explican muchos de los principios en los que se basa la Programación Estructurada clásica y la Programación Orientada a Objetos.

Cohesión

La cohesión indica hasta qué punto las tareas y responsabilidades de que consta un programa o parte de un programa están relacionados entre sí para conseguir un único fin , es decir, si el programa se centra en la solución de un solo problema. Bajo este punto de vista la cohesión puede ser fuerte o débil.

Cohesión fuerte

Cohesión fuerte indica que las responsabilidades y tareas de una pieza de código (un método, una clase, un componente o un programa) se relacionan entre sí con la intención de resolver un problema específico y común bien identificado. Esto es algo a lo que siempre se debe aspirar. **Cohesión fuerte es una característica típica de software de calidad.**

Cohesión fuerte en una clase

La cohesión fuerte, en una clase, indica que la clase define una sola entidad o concepto. Una entidad puede tener muchos roles (Pedro es un alumno, un empleado y un contribuyente). Cada uno de estos roles se definen con la misma clase. Una cohesión fuerte indicaría que la clase resuelve sólo uno de esos posibles roles, un problema, y no muchos al mismo tiempo.

Una clase encargada de hacer muchas cosas, es difícil de entender y mantener. Si se supone una clase que implementa un almacén de datos, proporciona funciones para la impresión, el envío de un correo electrónico y trabajar con funciones trigonométricas de una vez ¿Cómo podría llamarse esa clase? Si resulta difícil responder a esta pregunta, significa que se ha fracasado en lograr la cohesión fuerte y hay que separar la clase en varias más pequeñas y especializadas, cada una para solucionar una sola tarea.

Ejemplo: Cohesión fuerte en una clase

Un ejemplo de cohesión fuerte es la clase `java.lang.Math` que está especializada en una única funcionalidad: proporciona métodos para los cálculos matemáticos y constantes asociadas:

- `sin () , cos () , asin ()`
- `sqrt () , pow () , exp ()`
- `Math.PI , Math.E`

Cohesión fuerte en un método

Un método está bien escrito cuando realiza una sola tarea especializada y la ejecuta sin errores. Un método, que realiza tareas y operaciones relacionadas con diferentes funciones, tiene poca cohesión y debe ser desglosado en métodos más simples más cohesionados. Un ejemplo extremo que sirve para entender de lo que va el asunto es el siguiente: *¿Qué nombre debe tener un método que busca números primos, dibuja gráficos en 3D en la pantalla, se comunica con la red e imprime los registros extraídos de una base de datos?* Tiene escasa cohesión; y debería separarse de forma lógica en varios métodos. **Poder decidir un nombre inequívoco y concreto para un método es indicador de buena cohesión funcional.**

Cohesión débil

Una cohesión débil se observa en los métodos que realizan varias tareas no relacionadas funcionalmente. Los métodos poco cohesionados suelen tener varios grupos diferentes de parámetros, con el fin de realizar las diferentes tareas. Requieren datos no relacionados entre sí, de manera poco natural o forzada. **La cohesión débil es perjudicial y debe evitarse.**

Ejemplo: Cohesión débil
Una clase que tiene cohesión débil:

```
public class Magica {
    public void imprimirDocumento(Documento d) { ... }
    public void enviarEmail(String contenedor,
        String asunto, String texto) { ... }
    public void calcularDistanciaEntrePuntos(
        int x1, int y1, int x2, int y2) { ... }
}
```

Buenas prácticas de cohesión

Mantener una cohesión fuerte es la forma más eficaz para escribir código limpio o de calidad. Cada concepto se asocia de manera sencilla a una sola parte o módulo de código fuente. El código resultante es más fácil de mantener y reutilizar.

Por el contrario, **con una cohesión débil, cada cambio** es una bomba de relojería, ya que **podría afectar a otras funciones.** A veces una tarea lógica se extiende a varios módulos diferentes; cualquier cambio se hace más costoso y difícil. **La reutilización de código también se dificulta, porque un componente hace varias tareas no relacionadas y la reutilización exige condiciones idénticas que rara vez vuelven a repetirse.**

Acoplamiento

El acoplamiento describe principalmente la medida en que los componentes / clases dependen uno del otro. Se puede hablar de acoplamiento débil y acoplamiento fuerte. **El acoplamiento débil, por lo general, se correlaciona con una cohesión fuerte y viceversa.**

Acoplamiento débil

Se dice que un elemento de código (programa / módulo / clase / método) tiene un acoplamiento débil si para comunicarse con otro elemento de código utiliza un interfaz bien definido y mínimo. Un cambio en la implementación de un componente con poco acoplamiento no se refleja en los demás con los que se comunica.

Cuando se escribe código fuente, no se deben tener en cuenta las características internas de los componentes que no sean expresadas a través de su interfaz.

La interfaz de cualquier elemento o pieza de código tiene que ser simplificado al máximo y definir sólo el comportamiento que se requiere para el trabajo del componente ocultando

todos los detalles innecesarios.

Un acoplamiento débil es una característica clave a la que se debe aspirar. Es una de las cualidades para una programación limpia de calidad.

Ejemplo: Acoplamiento débil
Se da una independencia funcional entre las clases y entre los métodos:

```
class Informe {
    public boolean leerFichero(String fichero) {...}

    public boolean guardarAFichero(String fichero) {...}
}

class Impresora {
    public static int imprimir(Informe inf) {...}
}

class Ejemplo {
    public static void main(String[] args) {
        Informe miInforme = new Informe();
        miInforme.leerFichero("InformeDiario.xml");
        Impresora.imprimir(miInformet);
    }
}
```

En este ejemplo, ninguno de los métodos depende de los otros. Los métodos se basan sólo en los parámetros, que se les pasan. Si los métodos necesitaran utilizarse en otros programas, no tendrían ningún problema en funcionar correctamente.

Acoplamiento fuerte

Un acoplamiento fuerte se produce:

- **Cuando hay muchos parámetros de entrada y de salida.**
- **Cuando se usan características de otro componente de manera no documentada** (por ejemplo, una dependencia de los atributos estáticos en otra clase).
- **Cuando se usan muchos parámetros de control que indican el comportamiento de los datos reales.**

Un acoplamiento fuerte entre dos o más métodos, clases o módulos; **significa que no pueden trabajar independientemente uno del otro y que un cambio en uno de ellos también afectará el resto.** Esto conduce a un **código difícil de seguir y causa grandes dificultades con su mantenimiento.**

Ejemplo: Acoplamiento fuerte
Se da una dependencia funcional fuerte entre las clases y entre los métodos:

```
class ParametrosMat {
```

```
    public static double operando;
    public static double resultado;
}

class UtilMat {
    public static void sqrt() {
        ParametrosMat.resultado = calcSqrt(ParametrosMat.operando);
    }
}

class Trasbordador {
    public static void main(String[] args) {
        ParametrosMat.operando = 64;
        UtilMat.calcSqrt();
        System.out.println(ParametrosMat.resultado);
    }
}
```

El código de ejemplo es difícil de entender y mantener, y la probabilidad de errores al manipularlo, es alta.

¿Qué sucede si otro método llama a `calcSqrt()`, pasa sus parámetros a través de las mismas variables estáticas `operando` y `resultado`?

Si hay que usar la misma funcionalidad para obtener la raíz cuadrada en un programa posterior, no consistirá simplemente en copiar el método `calSqrt()`, sino que hay que copiar las clases **`ParametrosMat`** y **`UtilMat`** con todos sus métodos. Esto hace que el código sea difícil de reutilizar.

De hecho, el código anterior es un ejemplo de código malo de acuerdo a todas las reglas de la Programación Estructurada y Orientada a Objetos. Si se observa mejor, se pueden identificar varios defectos a los que se han hecho referencia anteriormente.

Buenas prácticas de acoplamiento

La manera más correcta y conveniente de invocar la funcionalidad de un componente es a través de una interfaz bien definida. De esta forma, si es necesario, la implementación de la funcionalidad puede ser sustituida sin afectar a los clientes (métodos) que lo usan. En terminología técnica se dice que se hace una ***programación con interfaces***.

Por lo general, ***una interfaz*** **describe y especifica un compromiso de implementación que cumple el módulo, la clase o componente.** Es una buena práctica de uso no confiar en cualquier otra cosa que no sea lo que está descrito expresamente en las interfaces. La utilización de las clases internas de un componente, que no son parte de la interfaz pública, no se recomienda debido a que puede ser sustituida, sin afectar al interfaz, que es a *lo que se ha comprometido en el contrato*.

Un buen ejemplo de cohesión fuerte y acoplamiento débil, se puede encontrar en la API de Java **`Collection`**. Estas clases que trabajan con colecciones, tienen una fuerte cohesión y cada una

resuelve un solo problema; permitiendo una fácil reutilización. El acoplamiento débil proporciona una alta calidad a su código. Las clases de aplicación de las colecciones no están relacionadas entre sí, cada una funciona a través de una interfaz estrictamente definida que no revela detalles de su implementación. Todos los métodos y atributos que no son de la interfaz están ocultos, con el fin de reducir la posibilidad de acoplamiento con ellos. Los métodos en las clases de **`Collection`** no dependen de variables estáticas y no se basan en otros datos que no sean los de su estado interno y los parámetros pasados. Esta es una experiencia de buena práctica que todos los programadores, antes o después, logran.

Código espagueti

El Código Spaghetti es un código no estructurado con una lógica confusa, difícil de leer, entender y mantener. Este es un código en el que la secuencia está distorsionada y poco clara. **Es un código que tiene una cohesión débil y acoplamiento fuerte**. Se utiliza la metáfora del espagueti, por motivos obvios. Tirando de un espagueti (es decir, una clase o método), todo el conjunto puede ir enredado tres él (es decir, cambiar un método o clase conduce a muchos más cambios debido a la fuerte relación entre ellos).

Un código espagueti es casi imposible de reutilizar, porque no se pueden separar en las distintas partes funcionales. Corresponde a uno de los antipatrones conocidos.

Cohesión y acoplamiento en otras disciplinas

Los principios de cohesión fuerte y acoplamiento débil se utilizan intensamente en otras ingenierías y disciplinas técnicas relacionadas con la construcción, la construcción de maquinaria, electrónica y muchos más campos.

Un ejemplo sencillo, una unidad de disco duro, ¿Realiza una sola función en el sistema? ¿No?, el disco duro sirve para almacenar datos. No enfría, no hace sonidos, no tiene capacidad de cálculo y no se utiliza como un teclado. Está conectado al ordenador con sólo dos cables, es decir, tiene una interfaz simple para el acceso y no está vinculado a otros periféricos. El disco duro funciona por separado y otros dispositivos no están preocupados acerca de cómo funciona exactamente. La CPU le manda leer y lee, le manda a escribir y escribe. ¿Cómo lo hace exactamente? no hay porqué saberlo para utilizarlo, permanece oculto dentro de él. Los diferentes modelos pueden funcionar de diferentes maneras, pero eso es una cuestión interna.

Se puede ver que un ordenador tiene una fuerte cohesión, acoplamiento mínimo, buena abstracción y bastante encapsulación.

Así es como se debe poner en práctica la escritura de unidades o piezas de programación:

- **Una pieza o unidad de código debe hacer sólo una cosa y ser pequeña.**
- **Hacerlo bien, con un diseño interno claro, sencillo y, si es necesario, optimizado.**
- **No depender de otras unidades o partes, o hacerlo mínimamente.**
- **Tener una interfaz concreta, clara y única.**
- **Representar fielmente y con naturalidad la realidad del problema a resolver.**
- **Ocultar los detalles de su funcionamiento interno.**

Otro ejemplo:

¿Qué sucedería, si el procesador, el disco duro, la unidad de CD-ROM y el teclado se sueldan, sin conectores, a la placa base del ordenador? Esto significa que si cualquier parte se estropea, habría que tirar todo el equipo. Se puede comprobar cómo el hardware no puede funcionar bien con un acoplamiento excesivo y cohesión débil. Lo mismo se aplica para el software.

Paquetes Java

Cláusula package

Un *paquete* es un conjunto de clases con algún nivel de cohesión y afinidad. Equivale al concepto de librería existente en otros lenguajes o sistemas. **las clases pueden definirse como pertenecientes a un paquete y pueden usar otras clases definidas en ese u otros paquetes**.

Los paquetes delimitan el *espacio de nombres* (*namespace*). **El nombre de una clase debe ser único dentro del paquete donde se define. Dos clases con el mismo nombre en dos paquetes distintos pueden coexistir e incluso pueden ser usadas en el mismo programa.**

Una clase se declara perteneciente a un paquete con la cláusula `package`, cuya sintaxis es:

```
package nombre_package;
```

La cláusula `package` debe ser la primera sentencia de un archivo fuente Java. Cualquier clase declarada en ese archivo pertenece al paquete indicado.

Por ejemplo, un fichero que contenga el código fuente de una clase pública puede indicar que ésta está incluida en un paquete, con algo parecido a lo siguiente:

```
package unPaquete;

//. . .

public class UnaClase {

	//. . .

}
```

La cláusula `package` es opcional. Si no se utiliza, las clases declaradas en el archivo fuente pertenecen al paquete por defecto; sin nombre.

La agrupación de clases en paquetes contribuye a mejorar la organización de los programas, manteniendo las clases relacionadas que cooperan en ubicaciones comunes. También influye en la configuración del control de acceso a los miembros de las clases.

Cláusula import

Cuando se referencia cualquier clase dentro de otra se asume, si no se indica otra cosa, que ésta otra está declarada en el mismo package. Por ejemplo:

```
package geometria;

//...

public class Circulo {

	// atributos
	Punto centro;

	//...
}
```

En esta declaración se crea la clase **`Circulo`** perteneciente al paquete **`geometria`**. Esta clase usa la clase **`Punto`**. Java asume que **`Punto`** pertenece también al paquete **`geometria`**, y tal como está hecha la definición, para que la clase **`Punto`** sea accesible, es necesario que esté definida en el mismo paquete.

Si la clase **`Punto`** no estuviera dentro del mismo paquete, sería necesario hacer accesible el espacio de nombres donde está definida. Esto se hace con la cláusula **`import`**. Por ejemplo si la clase **`Punto`** estuviera definida de esta forma:

```
package geometriaBase;

//...

class Punto {
	// atributos
	int x;
	int y;

	//...
}
```

Entonces, para usar la clase **`Punto`** en la clase **`Circulo`** debería poner:

```
package geometria;

```

```
import geometriaBase.Punto; // Hace accesible la clase Punto
//...

class Circulo {

	// atributos
	Punto centro;

	//...
}
```

Opcionalmente también se puede escribir **`import GeometriaBase.*;`** para hacer accesibles todos los nombres (todas las clases) declaradas en el paquete **`geometriaBase`**.

Por otro lado, también es posible hacer accesibles los nombres de un paquete sin usar la cláusula **`import`** calificando completamente los nombres de aquellas clases pertenecientes a otros paquetes. Por ejemplo:

```
package geometria;
//...

public class Circulo {

	// atributos
	geometriaBase.Punto centro;

	//...
}
```

Si no se usa **`import`** es necesario especificar el nombre del paquete cada vez que se usa el nombre **`Punto`**.

La cláusula **`import`** tiene las siguientes características:

- Indica al compilador de dónde debe buscar clases adicionales si no puede encontrarlas en el paquete actual.
- Delimita el acceso a los espacios de nombres.
- No incluye o copia código fuente u objeto alguno en el fichero fuente donde aparece.
- En una clase puede haber tantas sentencias **`import`** como sean necesarias.
- Las cláusulas **`import`** se colocan después de la cláusula **`package`** (si es que existe) y antes de las definiciones de las clases.
- No pueden ser importadas las clases del paquete por defecto.

Nombres de paquetes

Los paquetes se pueden nombrar usando nombres compuestos separados por puntos, de forma similar a como se componen las direcciones URL de Internet. Por ejemplo se puede tener un

paquete de nombre:

`misPaquetes.geometria.base`

Cuando se utiliza esta estructura se habla de paquetes y *subpaquetes*. En el ejemplo, `misPquetes` es el *paquete base*, geometria es un subpaquete de `misPaquetes` y `base` es un subpaquete de `geometria`.

De esta forma se pueden tener los paquetes organizados según una jerarquía equivalente a un sistema de archivos jerárquico.

La API de java está estructurada de esta forma, con un primer calificador (`java` o `javax`) que indica la base, un segundo calificador (`awt,util,swing`, etc.) que indica el grupo funcional de clases y opcionalmente subpaquetes en un tercer nivel, dependiendo de la amplitud del grupo. Cuando se crean paquetes de usuario no es recomendable usar nombres de paquetes que empiecen por `java` o `javax`.

Ubicación de paquetes en el sistema de ficheros

Además del significado lógico descrito hasta ahora, los paquetes también tienen un significado *más físico* que sirve para almacenar los módulos ejecutables (ficheros con extensión `.class`) en el sistema de archivos del ordenador.

Si se define una clase de nombre **`miClase`** que pertenece a un paquete de nombre `misPaquetes.geometria.base`. Cuando la JVM vaya a cargar en memoria **`miClase`** buscará el módulo ejecutable (de nombre `miClase.class`) en un directorio en la ruta de acceso `misPackages/Geometria/Base`. Esta ruta deberá existir y estar accesible a la JVM para que encuentre las clases.

Si una clase no pertenece a ningún paquete (no existe cláusula **package**) se asume que pertenece a un paquete por defecto sin nombre, y la JVM buscará el archivo `.class` en el directorio actual.

Para que una clase pueda ser usada fuera del paquete donde se definió debe ser declarada con el modificador de acceso **`public`**, de la siguiente forma:

```
package geometriaBase;

//...

public class Punto {
	// atributos
	int x;
	int y;

	//...
}
```

Si una clase no se declara **`public`** sólo puede ser usada por clases que pertenezcan al mismo paquete.

Modelado orientado a objetos

Cuando hay que resolver un problema con software, por lo general, ese problema existe en el mundo real y es necesario comprenderlo y acotarlo antes de diseñar y desarrollar una solución software.

El *modelado orientado a objetos* (*MOO*) es un proceso asociado a la *POO* donde se identifican todos los objetos relacionados con el dominio del problema que se está solucionando (se crea un modelo). Se capturan sólo las clases características, que son importantes para la resolución de este problema; el resto se ignoran. De esta manera, **se crea una representación simplificada de la realidad (modelo), de tal manera que permite resolver el problema de manera más simple y limitada.**

Por ejemplo, si se modela un *sistema de gestión de viajeros*, habrá que representar las características más importantes de un pasajero: nombre, edad, si tiene algún descuento, etc. Un pasajero tiene muchas otras características no importantes que no son relevantes para la gestión que se pretende, como el color de sus ojos, qué número de calzado gasta, qué libros le gustan o la cerveza que bebe, etc.

El modelado permite crear una representación simplificada de la realidad con el fin de resolver una tarea específica.

En el modelado orientado a objetos, el modelo se crea por medio de un análisis orientado a objetos: a través de las clases, atributos de clase, métodos de clase, los objetos, las relaciones entre las clases, etc.

El modelado pone en juego procesos mentales complejos relacionados con las capacidades de análisis y síntesis combinadas. Se ayuda de metodologías, técnicas y herramientas específicas.

Pasos en el modelado orientado a objetos

El modelado orientado a objetos se realiza generalmente en los siguientes pasos:

- Identificación de clases.
- Identificación de los atributos de clase.
- Identificación de las operaciones en clases.
- Identificación de las relaciones entre clases.

Identificación de clases

Supóngase que se tiene el siguiente extracto de la especificación de un sistema:

"El usuario debe ser capaz de describir cada producto según sus características, incluyendo el nombre y número de producto. Si el código de barras no coincide con el producto, se debe generar un error en la pantalla. Tiene que haber un informe diario para todas las transacciones especificadas en el apartado 9.3"

Se identifican los conceptos clave que se consideran clave:

*"El **usuario** debe ser capaz de describir cada **producto** según sus **características**, incluyendo el **nombre** y **número de producto**. Si el **código de barras** no coincide con el producto, se debe generar un **error** en la **pantalla**. Tiene que haber un **informe diario** para todas las **transacciones** especificadas en el apartado 9.3"*

Se han identificado, esencialmente, las clases que se necesitarán. Los nombres de las clases son los sustantivos en el texto, por lo general los nombres comunes en singular, como por ejemplo: estudiante, mensaje, león, etc. Se deben evitar los nombres que no aparecen en el texto.

A veces es difícil determinar si algún concepto, tema o fenómenos del mundo real tiene que ser incluido como una clase en el modelo. Por ejemplo, la dirección puede ser definida como una clase *Direccion* o un String. Cuanto más se conozca el problema, más fácil será decidir qué entidades deben ser representadas como clases.

Cuando una clase se hace grande y complicada tiene que ser descompuesta en varias más pequeñas y manejables. Después se acoplan por *uso, agregación, composición* o *herencia*.

Identificación de los atributos

Las clases tienen atributos (características), por ejemplo, la clase **`Estudiante`** tiene un `nombre`, y una lista de `cursos`. No todas las características son importantes para un sistema de software. Por ejemplo, en cuanto a la clase **`Estudiante`** se refiere, el color de los ojos es una característica no esencial. Sólo las características esenciales deben ser representadas (modeladas).

Identificación de las operaciones

Cada clase tiene una serie de responsabilidades claramente definidas:

- Qué objetos o procesos del mundo real que identifica
- Qué tareas realiza.

Cada acción en el programa se lleva a cabo por uno o varios métodos en alguna clase. Las acciones se modelan como operaciones (métodos).

Para establecer el nombre de un método se recomienda utilizar una combinación de *verbo* + *sustantivo*, por ejemplo: `imprimirInforme()`, `conectarABaseDatos()`. Normalmente no se podrán definir todos los métodos de una clase de manera inmediata; en primer lugar, se identifican los métodos más importantes, los que implementan las responsabilidades básicas de la clase. Con el tiempo aparecen otros métodos.

Identificación de las relaciones entre las clases

Si un **`Estudiante`** pertenece a una facultad y esto es importante para la tarea que se están solucionando, como por ejemplo, *hay que obtener qué estudiantes están matriculados en una materia*, los estudiantes deben aparecer en una lista de la clase **`Facultad`**. Estas relaciones se llaman asociaciones.

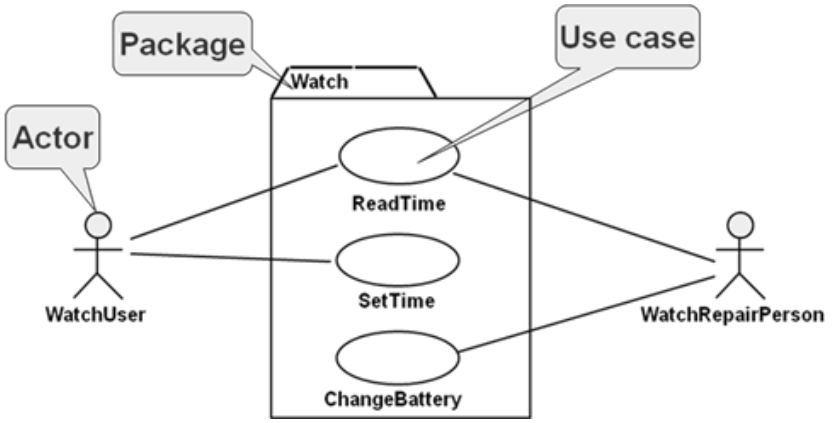
Notación UML

La notación UML define varios tipos de diagramas que facilitan la representación gráfica del modelo desde distintos puntos de vista y finalidad.

Diagramas de casos de uso

Se utilizan para la captura de requisitos y la descripción de las posibles acciones. Los Actores realizan papeles (tipos de usuarios) cuando interactúan con el sistema que se está modelando. Los casos de uso describen la interacción entre los actores y el sistema. El modelo de casos de uso es un conjunto de especificaciones en forma de ficha que proporcionan una descripción completa de la funcionalidad de un sistema.

Ejemplo: Diagrama de casos de uso



Un actor representa un agente que interactúa con el sistema (un usuario o un sistema externo). El actor tiene un nombre único y una descripción. Por ejemplo, `WatchUser` y la `WatchRepairPerson`.

Un caso de uso (los círculos o elipses) describe una funcionalidad única del sistema, una sola acción que puede realizarse por un actor. Tiene un nombre único y se relaciona con los actores. Puede tener condiciones de entrada y de salida. Representa un flujo de operaciones de un proceso y puede tener requisitos adicionales. Hay tres casos de uso en el diagrama de ejmplo anterior: `ReadTime`, `SetTime` y `ChangeBattery`.

Un paquete contiene varios casos de uso relacionados lógicamente.
Las líneas conectan a los actores con los casos de uso que realizan. Un actor puede realizar o estar

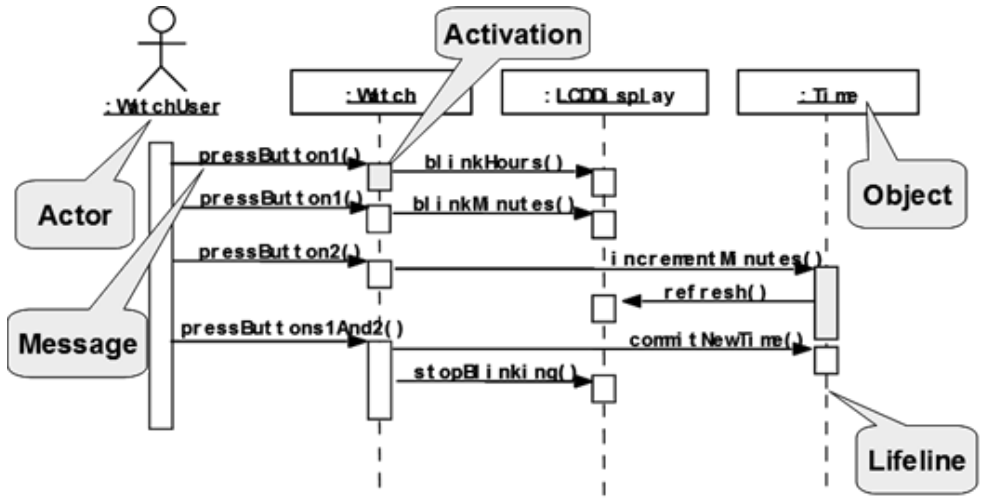
involucrado en uno o varios casos de uso.

Diagramas de secuencia

Los diagramas de secuencia se utilizan cuando se modelan los requisitos de la especificación del proceso y la descripción amplia de los escenarios de casos de uso. Permiten describir a los participantes adicionales en los procesos y la secuencia de las acciones en el tiempo. Se utilizan en el diseño y las descripciones de las interfaces del sistema.

Los diagramas de secuencia describen lo que sucede a lo largo del tiempo. Las interacciones y las secuencia de pasos proporcionan una visión dinámica del sistema, algo parecido a un algoritmo.

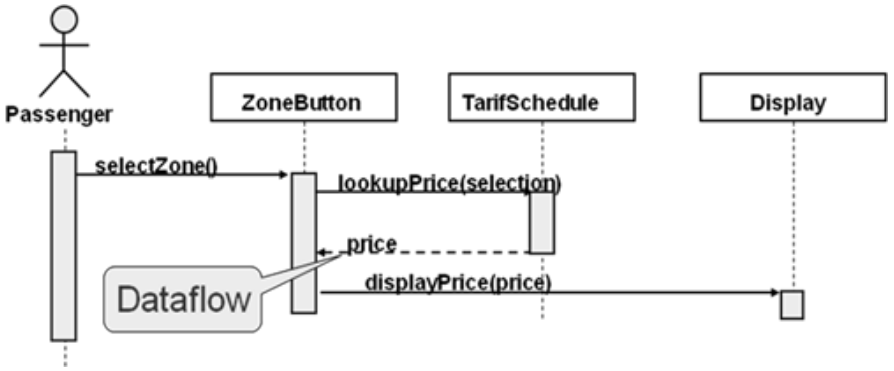
Ejemplo: Diagrama de secuencia



Las clases se representan con columnas (líneas de vida). Los mensajes (acciones) se representan con flechas horizontales con el nombre encima. Los objetos participantes se representan con rectángulos horizontales conectados con línea discontinua a su clase. El período de actividad de una clase se representa en forma de rectángulos verticales discontinuos correspondientes a la activación de un determinado mensaje (método).

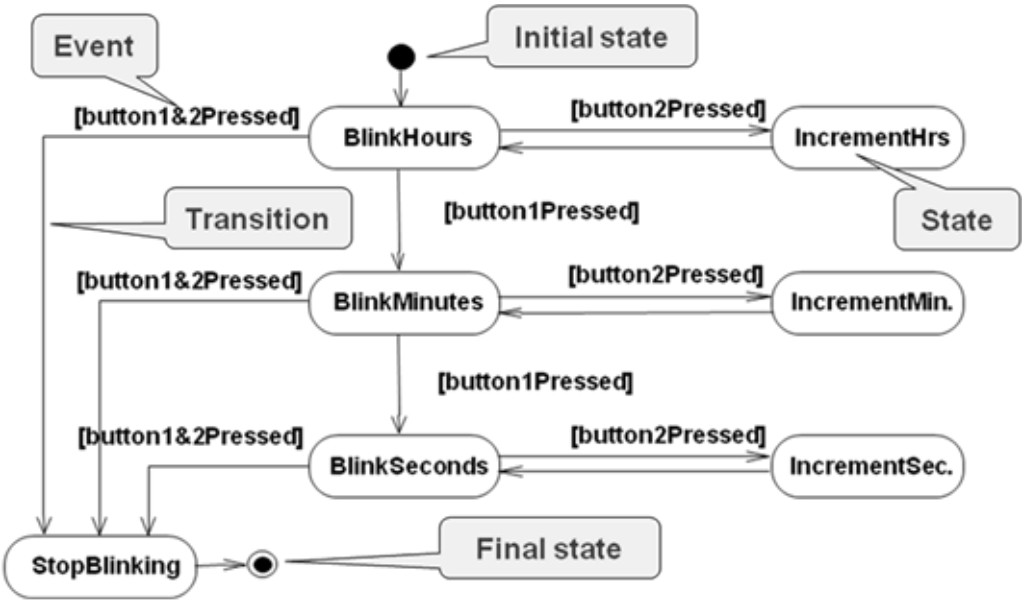
Ejemplo: Mensajes

La dirección de la flecha designa el remitente y el destinatario de un mensaje (una llamada a un método en programación orientada a objetos). Las líneas horizontales discontinuas representan el flujo de datos:



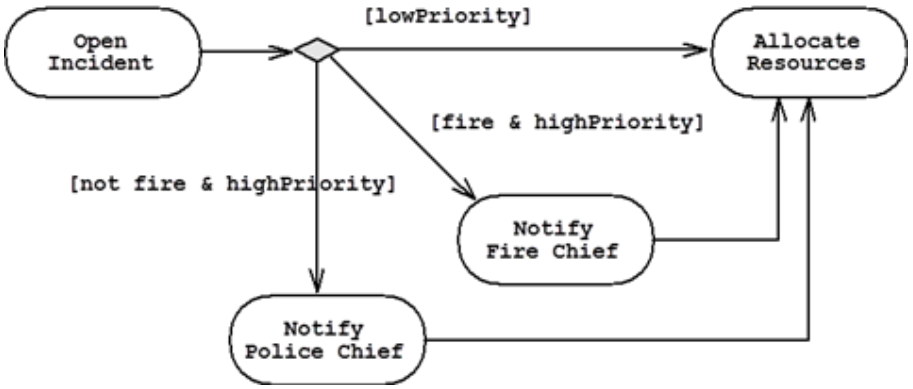
Diagramas de estado

Los diagramas de estado describen los posibles estados de cierto proceso y las posibles transiciones entre ellos, junto con las condiciones de las transiciones. Representan autómatas de estados finitos(máquinas de estado). A continuación se puede ver un ejemplo de diagrama de estado que ilustra los estados y las transiciones de un proceso típico para cambiar la hora actual de un reloj que tiene dos botones y una pantalla:



Diagramas de actividad

Los diagramas de actividades son un tipo especial de diagramas de estado donde las condiciones son las acciones. Muestran el flujo de las acciones en un sistema:



Patrones de diseño

Historia [VIDAL1]

El término patrón fue utilizado por primera vez por el arquitecto Christopher Alexander en el libro "A Pattern Language: Towns, Buildings, Construction", donde definió una serie de patrones arquitectónicos. Alexander define:

"Un patrón describe un problema que ocurre a menudo, acompañado por un intento de solución para el problema."

Christopher Alexander, 1977

En 1987, Ward Cunningham y Kent Beck estaban trabajando con Smaltalk, diseñando interfaces de usuario. Para ello, decidieron utilizar alguna de las ideas de Alexander y desarrollaron un pequeño lenguaje de patrones que serviría de guía a los programadores de Smaltalk. A partir de estas idea escribieron el libro "Using Pattern Languajes for Object-Oriented Programs".

Desde 1990 a 1994, Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (Gang of four) realizaron un primer catálogo de patrones de diseño. En 1994 publicarón el libro "Design Patterns – Elements of Reusable Object-Oriented Software" que introducía el término de *patrón de diseño* en el desarrollo del software.

Los patrones de diseño aparecen como soluciones probadas y altamente eficientes a los problemas más comunes de modelado y programación orientado a objetos.

Noción de patrón de diseño [VIDAL1]

Un patrón de diseño es una solución repetible a problemas típicos y recurrentes en el diseño del software. Son soluciones basadas en la experiencia y que se ha demostrado que funcionan. No son un diseño terminado que puede traducirse directamente a código, sino más bien una descripción sobre cómo resolver el problema, la cual puede ser utilizada en diversas situaciones. Eric Gamma define los patrones de diseño como:

"Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí para resolver un problema de diseño general en un contexto particular."

En general, un patrón de diseño consta de cuatro elementos esenciales:

1. El nombre del patrón se utiliza para describir un problema de diseño, su solución y consecuencias en una o dos palabras. Nombrar un patrón incrementa inmediatamente nuestro vocabulario de diseño. Esto nos permite diseños a un alto nivel de abstracción. Tener un vocabulario de patrones nos permite hablar sobre ellos con nuestros compañeros, en nuestra documentación, e incluso a nosotros mismos.
2. El problema describe cuándo aplicar el patrón. Se explica el problema y su contexto. Esto

podría describir problemas de diseño específicos tales como algoritmos como objetos. Podría describir estructuras de clases o objetos que son sintomáticas de un diseño inflexible. Algunas veces el problema incluirá una lista de condiciones que deben cumplirse para poder aplicar el patrón.

3. La solución describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. La solución no describe un diseño particular o implementación, porque un patrón es como una plantilla que puede ser aplicada en diferentes situaciones. En cambio, los patrones proveen una descripción abstracta de un problema de diseño y como una disposición general de los elementos (clases y objetos en nuestro caso) lo soluciona.
4. Las consecuencias son los resultados de aplicar el patrón. Estas son muy importantes para la evaluación de diseños alternativos y para comprender los costes y beneficios de la aplicación del patrón.

Especificación de los Patrones de Diseño [VIDAL1]

La documentación de un patrón de diseño describe el contexto en el que se utiliza el patrón, las fuerzas dentro del contexto en el que el patrón busca resolver, y la solución sugerida. No existe un formato único para la documentación de los patrones de diseño. El formato más frecuentemente utilizado es el utilizado por GoF en su libro de Patrones de Diseño. Contiene las siguientes secciones:

- ❏ **Nombre del patrón:** identifica al patrón.
- ❏ **Clasificación del patrón:** creacional, estructural o de comportamiento.
- ❏ **Intención:** ¿Qué problema pretende resolver el patrón?
- ❏ **También conocido como:** Otro nombres por los que es conocido el patrón.
- ❏ **Motivación:** Escenario de ejemplo para la aplicación del patrón.
- ❏ **Aplicabilidad:** Usos comunes y criterios de aplicabilidad del patrón.
- ❏ **Estructura:** Diagramas de clases oportunos para describir las clases que intervienen en el patrón.
- ❏ **Participantes:** Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón.
- ❏ **Colaboraciones:** Explicación de las interrelaciones que se dan entre los participantes.
- ❏ **Consecuencias:** Consecuencias positivas y negativas en el diseño derivado de la aplicación del patrón.
- ❏ **Implementación:** Técnicas o comentarios oportunos de cara a la implementación del patrón.
- ❏ **Código de ejemplo:** Código fuente de ejemplo de la implementación del patrón.
- ❏ **Usos conocidos:** Ejemplos de sistemas reales que usan el patrón.
- ❏ **Patrones relacionados:** Referencias cruzadas con otros patrones.

Objetivos de los patrones de diseño

- ❖ **Proporcionar catálogos de elementos reusables en el diseño de sistemas software.**
- ❖ **Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.**
- ❖ **Formalizar un vocabulario común entre diseñadores.**
- ❖ **Estandarizar el modo en que se realizan los diseños.**

- **Facilitar el aprendizaje a las nuevas generaciones de diseñadores. Condensa el conocimiento ya existente.**

Asimismo, no pretenden:

- Imponer ciertas alternativas de diseño frente a otras.
- Eliminar la creatividad inherente al proceso de diseño.

No es obligatorio utilizar los patrones, sólo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable. **Abusar o forzar el uso de los patrones puede ser un error**.

Seleccionar un patrón de diseño [VIDAL1]

De entre los diferentes patrones de diseño que existen puede ser complicado la elección de uno para el diseño de una aplicación. Entre las pautas que hay que seguir a la hora de seleccionar un patrón se pueden indicar las siguientes:

1. Observar como el patrón de diseño resuelve los problemas de diseño.
2. Revisar las secciones de “Objetivo”.
3. Estudiar la interrelación entre los diferentes patrones. Estudiar estas relaciones puede guiar directamente al patrón correcto o grupo de patrones.
4. Estudiar los patrones con un propósito similar. Hay patrones que son muy parecidos estudiar sus similitudes y sus diferencias te ayudará en la elección del patrón que mejor se adapte a tu problema.
5. Examinar las razones de cambio. Mirar las causas de rediseño. Luego mirar los patrones que te ayudan a evitar las causas de rediseño.
6. Considerar lo que se debería cambiar en el diseño concreto. Este apartado es el opuesto al anterior, es decir al enfocado sobre las razones de cambio. En lugar de considerar que podría forzar un cambio en un diseño, considera lo que quieres que Guía de construcción de software en Java con patrones de diseño sea capaz de cambiar sin rediseñar. El objetivo aquí es encapsular el concepto que varía, un fin de muchos patrones de diseño. Son aspectos del diseño que los patrones de diseño permiten que varíen independientemente, por lo tanto puedes hacer cambios sin rediseñar.

¿Cómo usar un patrón de diseño?

Una vez seleccionado el patrón de diseño que se va a aplicar hay que tener en cuenta las siguientes claves para utilizar el patrón:

- Leer el patrón de diseño por encima. Prestar una atención particular a las secciones “Aplicabilidad” y “Consecuencias” para asegurarse de que el patrón es correcto para tu problema.
- Observar la estructura, los elementos que participan en el patrón y las colaboraciones entre ellos. Asegurarse de que comprendes las clases y objetos del patrón y cómo se relacionan.
- Mirar la sección “Código del ejemplo” para ver un ejemplo concreto del patrón en código. Estudiar el código te enseñará cómo implementar el patrón.
- Escoger nombres significativos de los elementos que participan en el patrón para el

contexto de la aplicación. Los nombres de los elementos de un patrón son normalmente demasiado abstractos para aparecer directamente en una aplicación. No obstante, es muy útil incorporar los nombres de los participantes en el nombre que aparece en la aplicación. Esto te ayudará a tener el patrón más explícito en la implementación.

- Definir las clases. Declarar sus interfaces, establecer sus relaciones de herencia, y definir las variables instanciadas que representan datos y referencias de objetos.
- Identificar las clases existentes en tu aplicación que el patrón afectará y modificará adecuadamente.
- Definir nombres específicos para las operaciones en dicha aplicación. Otra vez, los nombres dependen generalmente de la aplicación. Utilizar las responsabilidades y colaboraciones asociadas con cada operación como guía.
- Implementar las operaciones que conllevan responsabilidad y colaboración en el patrón. La sección “Implementación” te ofrece una guía en la implementación. Los ejemplos de la sección “Código del ejemplo” también te ayudan.

Categorías de patrones

Según la escala o nivel de abstracción:

Patrones de arquitectura

Son aquellos que expresan un esquema organizativo estructural fundamental para sistemas de software.

Patrones de diseño

Son aquellos que expresan esquemas para definir estructuras de diseño (o sus relaciones) con las que construir sistemas de software.

Dialectos

Son patrones de bajo nivel específicos para un lenguaje de programación o entorno concreto.

Antipatrones de diseño

Son esquemas con forma semejante a la de un patrón que intentan ilustrar y prevenir contra errores comunes de diseño en el software. La idea de los antipatrones es dar a conocer los problemas, muy frecuentes, que acarrean ciertos diseños para intentar evitar que los sistemas acaben una y otra vez en el mismo callejón sin salida por haber cometido los mismos errores típicos.

Patrones creacionales

Corresponden a patrones de diseño software que **solucionan problemas de creación de instancias**. Ayudan a encapsular y abstraer dicha creación los más habituales son:

Object Pool

El patrón *Batería de Objetos*, no pertenece a los patrones especificados por *GoF* (Grupo de los

cuatro), con este patrón **se obtienen objetos nuevos a través de la clonación**.

Se utiliza cuando el coste de crear un nuevo objeto de una clase, desde cero, es mayor que el de clonarlo. Especialmente útil con objetos muy complejos. Se especifica un tipo de objeto a crear y se utiliza una interfaz del prototipo para crear un nuevo objeto por clonación. El proceso de clonación se inicia instanciando un nuevo objeto de la clase que queremos clonar.

Abstract Factory

El patrón *Fábrica Abstracta*, permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí; haciendo transparente el tipo de familia concreta que se esté usando. El problema a solucionar por este patrón es el de crear diferentes familias de objetos, como por ejemplo la creación de interfaces gráficas de distintos tipos (ventana, menú, botón, etc.).

Builder

El patrón *Constructor Virtual*, abstrae el proceso de creación de un objeto complejo, centralizando dicho proceso en un único punto.

Factory Method

El patrón *Método Factoría*, es otro patrón de diseño muy común. **La construcción de un objeto no se realiza directamente, sino por un método delegado que lo crea.** Esto permite que el método de fábrica decida qué instancia específica se necesita para el uso particular que se dará al objeto creado. La solución puede depender del entorno, de un parámetro o de algún ajuste del sistema.

Este patrón cuando se aplica, al máximo nivel de genericidad, se basa en una clase creadora abstracta de la que se deriva otra específica que implementa el método factoría. Parte del principio de que las subclases creadoras especializan los métodos factoría.

Ejemplo 1: Método factoría

Un método factoría encapsula la creación de objetos. Esto es útil si el proceso de creación es muy complicado, depende del entorno, depende archivos de configuración o depende de datos de entrada por parte del usuario.

```
public final class Integer
        extends Number
        implements Comparable<Integer> {

    // ...

    public static Integer valueOf(String s)
            throws NumberFormatException {
        return new Integer(parseInt(s, 10));
    }

    // ...
}
```

El método `valueOf(String)` produce un ejemplar (número) a partir del texto. También hay un parámetro -el número 10-, que indica a qué notación se espera para el número en la cadena de texto.

Ejemplo 2: Método factoría

```
public abstract class Calendar implements Serializable,/*...*/ {

	// ...

	public static Calendar getInstance() {
		Calendar cal = createCalendar(TimeZone.getDefaultRef(),
					Locale.getDefault());
		cal.sharedZone = true;
		return cal;
	}

	private static Calendar createCalendar(TimeZone zone, Locale aLocale) {

	// Si el entorno local especificado no es de Tailandia,
	// devuelve una instancia del calendario budista
	if ("th".equals(aLocale.getLanguage())
					&& "TH".equals(aLocale.getCountry()))) {

		return new sun.util.BuddhistCalendar(zone, aLocale);
	}
	else if ("JP".equals(aLocale.getVariant())
					&& "JP".equals(aLocale.getCountry())
					&& "ja".equals(aLocale.getLanguage())) {

		return new JapaneseImperialCalendar(zone, aLocale);
	}

		// De lo contrario crear el calendario predeterminado
		return new GregorianCalendar(zone, aLocale);
	}

	// ...
}
```

Se puede suponer que ambos métodos `getInstance()` y `createCalendar()` son métodos de fábrica. El método `getInstance()` Obtiene la *configuración regional* y la *zona horaria* y crea una instancia con esa información utilizando el segundo método de fábrica para crear una instancia real.

El método `CreateCalendar()` devuelve una instancia de la clase basada en la *configuración regional* y la *zona horaria* proporcionadas como parámetros. Devuelve uno de los tres objeto Calendar específicos posibles: **`BuddhistCalendar`**, **`JapaneseImperialCalendar`** o **`GregorianCalendar`**.

Prototype

El patrón *Prototipo*, crea nuevos objetos clonándolos de una instancia ya existente.

Singleton

El patrón *Instancia Única*, es uno de los patrones de diseño más conocido. Garantiza que una clase tenga una sola instancia u objeto durante su ejecución en un programa.
El patrón garantiza:

- La existencia de una instancia única, para la clase que aplica el patrón de diseño.
- La disponibilidad de un mecanismo de acceso global a dicha instancia u objeto.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que una clase controla el acceso a un recurso físico único (como puede ser el ratón, una impresora, o un archivo abierto en modo exclusivo). También es de aplicación cuando ciertos datos deben ser comunes, coherentes y estar disponibles para todos los demás objetos de una misma aplicación.

El patrón singleton se implementa básicamente en tres pasos:

1. Creando un **atributo estático** -referencia a la instancia única- de la propia clase que implementa el patrón.
2. Se configura en el **constructor privado**, o a través de otro método, una estructura de control condicional que crea el objeto sólo si no existe.
3. Se establece un **método estático** de acceso al atributo; la instancia única.

Para asegurar que la clase no puede ser utilizada para instanciar objetos con el operador **`new`** se configura el constructor con el cualificador de acceso **`private`** o **`protected`**.

La implementación del patrón Singleton en programas con múltiples hilos de ejecución puede verse comprometida si varios hilos de ejecución intentan crear la instancia al mismo tiempo y esta no existe todavía; **sólo uno de ellos debe lograr crear el objeto**. La solución clásica para este problema es utilizarla exclusión mutua en el método de creación de la clase que implementa el patrón.

Esquema, participantes y colaboraciones

El patrón singleton proporciona una *instancia única global* a las clases cliente, gracias a que:

- La propia clase es la responsable de crear su propia *instancia*.
- La clase dispone de un *constructor privado* para que no sea ejecutable directamente.
- Permite el acceso global a dicha instancia mediante un *método estático de clase de acceso a la instancia.*

Ejemplo: Singleton simple

```
public class AplicaSingleton {

	// Requerido por el patrón
	private static AplicaSingleton instancia = null;
```

```
	// Otros atributos, por ejemplo
	private int valor;

	/**
	 * Constructor por defecto de uso interno.
	 * Sólo se ejecutará una vez.
	 */
	private AplicaSingleton() {

		valor = 3;
	}

	/**
	 *  Método estático de acceso a la instancia única.
	 *  Si no existe la crea invocando al constructor interno.
	 *  Sólo se crea una vez; instancia única -patrón singleton-
	 *  @return instancia
	 */
	public static AplicaSingleton getInstancia() {
		if (instancia == null) {
			instancia = new AplicaSingleton();
		}
		return instancia;
	}

	/**
	 *  Método getValor().
	 *  Sólo estará disponible a través del objeto instancia.
	 *  @return valor
	 */
	public int getValor() {
		return valor;
	}

	/**
	 *  Método setValor().
	 *  Sólo estará disponible a través del objeto instancia.
	 *  @param v - El valor a cambiar
	 */
	public void setValor(int v) {
		valor = v;
	}
}
```

En el ejemplo se aprecia que:

- Hay un constructor privado con el fin de impedir instancias externas.
- Un atributo estático tiene asignada la única instancia.
- Se ejecuta una sola vez el constructor de la clase.
- Se utiliza la llamada *inicialización diferida* de la instancia, al momento del primer uso.
- El método para acceder a la instancia única se llama `getInstancia()`.

La versión simple del Singleton tiene varios problemas si se considera un escenario de programación concurrente; varios hilos de tarea pueden invocar cada uno por su cuenta al método

`getInstacia()`, con la posibilidad de que se produzcan varias instancias. Para prevenir este riesgo es necesario ejecutar el proceso de creación de la instancia de una manera exclusiva, asegurando realmente una única ejecución del constructor.

Ejemplo: Singleton concurrente

```
public class AplicaSingleton {

	// Requerido por el patrón
	private static AplicaSingleton instancia = null;

	// Otro atributo, a título de ejemplo
	private int valor;


	/**
	 * Constructor por defecto de uso interno.
	 * Sólo se ejecutará una vez.
	 */
	private AplicaSingleton() {

		valor = 3;
	}


	/**
	 *  Método de acceso a la instancia única.
	 *  Si no existe la crea invocando al constructor interno.
	 *  Sólo se crea una vez; instancia única -patrón singleton-
	 *  @return instancia
	 */
	public static Singleton getInstancia() {
		crearInstancia();
		return instancia;
	}


	/**
	 * Método auxiliar que utiliza la inicialización diferida
	 * sincronizada para protegerse de posibles concurrencias multi-hilo.
	 * Evita instanciación múltiple por concurrencia.
	 */
	private synchronized static void crearInstancia() {
		if (instancia == null) {
			instancia = new AplicaSingleton();
		}
	}

}
```

Se utiliza un método auxiliar estático, de uso interno, llamado `crearInstancia()`, encargado de crear la instancia. Es el que se marca como de uso exclusivo con **`synchronized`**.

Adicionalmente, se puede optimizar y mejorar la versión anterior utilizando sólo un *bloque*

sincronizado, no el método completo.

```
	/**
	 * Método auxiliar que utiliza la inicialización diferida
	 * sincronizada para protegerse de posibles concurrencias multi-hilo.
	 * activa la exclusión sincronizada sólo si no hay instancia creada.
	 * Evita instanciación múltiple por concurrencia.
	 */
	private static void crearInstancia() {
	   if (instancia == null) {
	       // Sólo se accede a la zona sincronizada
	       // cuando la instancia no está creada.
	       synchronized(AplicaSingleton.class) {
	           // En la zona sincronizada es necesario volver
	           // a comprobar que no se ha creado la instancia
	           if (instancia == null) {
	                   instancia = new AplicaSingleton();
	           }
	       }
	   }
	}
```

Ejemplo: Singleton concurrente con bloqueo de clonación

En Java existe otro riesgo que puede comprometer el requisito de *instancia única*; es el de la utilización del método estándar `clone()`, del que disponen todos los objetos Java por herencia de la clase **`Object`**. No se puede impedir la invocación y ejecución de `clone()`, lo único que se puede hacer es redefinirlo par que no haga nada y produzca una excepción.

```
public class AplicaSingleton {

	// Requerido por el patrón
	private static AplicaSingleton instancia = null;

	// Otro atributo, a título de ejemplo
	private int valor;


	/**
	 * Constructor por defecto de uso interno.
	 * Sólo se ejecutará una vez.
	 */
	private AplicaSingleton() {

		valor = 3;
	}


	/**
	 *  Método de acceso a la instancia única.
	 *  Si no existe la crea invocando al constructor interno.
	 *  Sólo se crea una vez; instancia única -patrón singleton-
```

```
         *  @return instancia
         */
        public static AplicaSingleton getInstancia() {
                crearInstancia();
                return instancia;
        }


        /**
         * Método auxiliar que utiliza la inicialización diferida
         * sincronizada para protegerse de posibles concurrencias multi-hilo.
         * activa la exclusión sincronizada sólo si no hay instancia creada.
         * Evita instanciación múltiple por concurrencia.
         */
        private static void crearInstancia() {
           if (instancia == null) {
                // Sólo se accede a la zona sincronizada
                // cuando la instancia no está creada.
                synchronized(AplicaSingleton.class) {
                    // En la zona sincronizada es necesario volver
                    // a comprobar que no se ha creado la instancia
                    if (instancia == null) {
                            instancia = new AplicaSingleton();
                    }
                }
           }
        }


        /**
         * Redefinición del método estándar clone().
         * Lanza una excepción si se intenta clonar la instancia única creada.
         * Evita instanciación múltiple por clonación utilizando el método
         * estándar heredado de Object.
         */
        @Override
        public Object clone() throws CloneNotSupportedException {

                throw new CloneNotSupportedException();
        }

}
```

El patrón *Singleton* presentado hasta ahora tiene todavía dos puntos débiles por donde podría comprometerse el requisito de *instancia única;* son debidos a otros mecanismos disponibles en los lenguajes, y en particular Java, que utilizados hábilmente conducen a tener *más de una instancia de la instancia única*, son la *serialización* y la *reflexión*. No se van a abordar aquí para no hacer demasiado extenso el capítulo. Para profundizar más, puede verse: http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-with-examples

Patrones arquitectónicos

Model View Controler (MVC)

El patrón *Modelo Vista Controlador*, es un patrón de arquitectura de software que separa los *datos* y la *lógica de negocio* de una aplicación, de la *interfaz de usuario* y el módulo encargado de gestionar los eventos y las comunicaciones. Este patrón plantea la separación del problema en tres capas:

- El *Modelo*.
  - Representa la realidad o dominio del problema a resolver por el software y su *lógica de negocio*.
  - Define los componentes para la representación de la información con la cual el sistema opera, determina dicha información y el comportamiento general de la aplicación, tanto en las consultas como en las actualizaciones.
  - Implementa los privilegios de acceso que se hayan descrito en las especificaciones de la aplicación (*lógica de negocio*).
  - Envía a la *Vista* la parte de la información que en cada momento se le solicita para que sea mostrada al usuario.
  - Las peticiones de acceso o manipulación de información llegan al *Modelo* siempre a través del *Controlador*.

- El *Controlador.*
  - Conoce los métodos y atributos del modelo.
  - Recibe y realiza lo que el usuario quiere hacer.
  - Define componentes para actuar de intermediario de la comunicación entre el *Modelo* y la *Vista*. Unifica la validación de las operaciones.
  - Responde a eventos y acciones del usuario y tramita peticiones al *Modelo.* (por ejemplo, cuando se quiere obtener la contraseña almacenada para compararla, durante un inicio de sesión).
  - También puede enviar comandos a la *Vista* asociada si se pide un cambio en la forma en que se presenta de *Modelo* (por ejemplo, desplazamiento o scroll por un documento o por los diferentes registros de una base de datos).

- La *Vista.*
  - Muestra o presenta un aspecto concreto del *Modelo* y es utilizada por el *controlador* durante la interaccionar con el usuario.
  - Define componentes para una solución concreta de interacción con el usuario.
  - Presenta la información y lógica de negocio definida en el *Modelo* con un formato adecuado para interactuar (*interfaz de usuario*) por lo tanto requiere de dicho *Modelo* la información que debe representar como salida.

Interacción de los componentes

Aunque se pueden encontrar diferentes implementaciones de MVC, el flujo de control que se sigue generalmente es el siguiente:

1. El usuario interactúa con la interfaz de usuario de la *Vista*, utilizando un objeto (por ejemplo, el usuario pulsa un botón, enlace, ejecuta un comando, etc.)

2. El *Controlador* recibe la notificación de la acción solicitada por el usuario. Un objeto controlador gestiona el evento que llega, frecuentemente a través de un gestor de eventos (handler) o callback.
3. El objeto controlador accede a algún objeto del *Modelo*, actualizándolo, posiblemente modificando algo de forma adecuada a la acción solicitada por el usuario (por ejemplo, el controlador actualiza el carro de la compra del usuario). Los controladores complejos están a menudo estructurados usando un patrón de comando que encapsula las acciones y simplifica su extensión.
4. El *Controlador* delega a los objetos de la *Vista* la tarea de desplegar la interfaz de usuario. La *Vista* obtiene sus datos del *Modelo* para generar la interfaz apropiada para el usuario donde se reflejan los cambios internos en el modelo (por ejemplo, produce un listado del contenido del carro de la compra). El *Modelo* no debe tener conocimiento directo sobre la *Vista*. Sin embargo, se podría utilizar el patrón Observador para proporcionar cierta retroinformación entre el *Modelo* y la *Vista*, permitiendo al *Modelo* notificar a los interesados de cualquier cambio. Un objeto de la *Vista* puede registrarse con el *Modelo* y esperar a los cambios, pero aun así el modelo en sí mismo sigue siendo independiente de la *Vista*. Este uso del patrón Observador no es posible en las aplicaciones Web puesto que las clases de la *Vista* están completamente desacopladas del *Modelo* y del *Controlador*. En general el *Controlador* no proporciona objetos del *Modelo* a la *Vista* aunque puede dar la orden a la *Vista* para que se actualice.
5. La interfaz de usuario espera nuevas interacciones del usuario, comenzando el ciclo nuevamente…

MVC y acceso a datos

Muchos sistemas informáticos utilizan un acceso a datos con distintas tecnologías incluso de bases de datos para gestionar los datos que debe utilizar la aplicación; en líneas generales del MVC, dicha gestión corresponde al *Modelo*. La unión entre *Capa de Presentación* y *Capa de Negocio* conocido en el paradigma de la Programación por capas representaría la integración entre la *Vista* y su correspondiente *Controlador* de eventos y *Capa de Acceso a datos.* MVC no pretende discriminar entre *Capa de Negocio* y *Capa de Presentación* pero si pretende separar la *Capa Visual Gráfica* de su correspondiente *programación y acceso a datos (Controlador)*, algo que mejora el desarrollo y mantenimiento de la *Vista* y el *Controlador* en paralelo, ya que ambos cumplen ciclos de vida muy distintos entre sí.

MVC en aplicaciones web

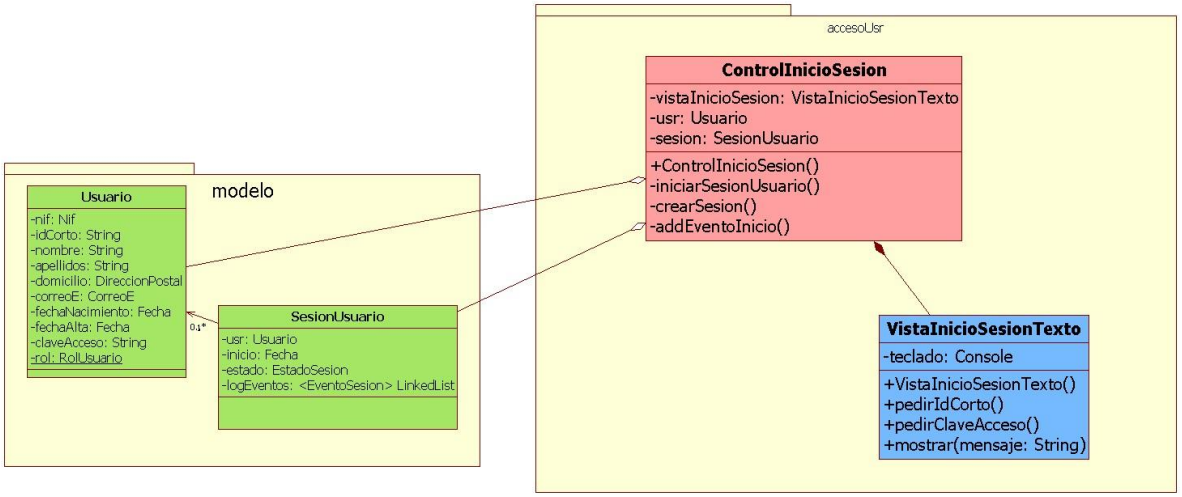
- Vista:
  - La página HTML
- Controlador:
  - Código que obtiene datos dinámicamente y genera el contenido HTML.
- Modelo:
  - la información almacenada en una base de datos o en XML junto con las reglas de negocio que transforman esa información (teniendo en cuenta las acciones de los usuarios)

MVC en el framework Java Swing

- Modelo:
  - El modelo lo realiza el desarrollador

- Vista:
  - Conjunto de objetos de clases que heredan de `java.awt.`**`Component`**
- Controlador:
  - El controlador es el *thread* de tratamiento de eventos, que captura y propaga los eventos a la *Vista* y al *Modelo.*
  - Clases de tratamiento de los eventos (a veces como clases anónimas) que implementan interfaces de tipo **`EventListener`** (**`ActionListener`**, **`MouseListener`**, **`WindowListener`**, etc.)

Ejemplo: MVC Inicio de sesión en modo texto



El controlador:

```
/**
 * MVC clase para el controlador de inicio de sesión en un sistema típico
 */

public class Controlador {

   private Vista vistaInicio;                          // Vista
   private Usuario usr;                                // Modelo
   private SesionUsuario sesion;                       // Modelo

   /**
    * Crea la vista y llama a sus métodos para interactuar
    */
   public Control() {
       vistaInicio = new Vista();
       iniciarSesionUsuario();
   }

   /**
    * Comprueba credenciales de usuario y
    * crea la sesión.
    */
   private boolean iniciarSesionUsuario() {

       String idCorto = vistaInicio.pedirIdCorto();
       String clave = vistaInicio.pedirClaveAcceso();
```

```
        // Busca el usuario del idCorto facilitado
        usr = Datos.getInstancia().buscarUsuario(idCorto);

        if (usr != null) {

            // Comprueba contraseña del usuario usr
            if (usr.claveOK(clave)) {
                crearSesion();
                addEventoSesionInicio();
                vistaInicioSesion.mostrar("Bienvenido, sesion iniciada...");
                return false;
            }
        }
        vistaInicioSesion.mostrar("Credenciales incorrectas...");
        //Excepcion credenciales incorrectas

        return true;
    }

    /**
     * Crea y asigna la sesión de usuario.
     */
    private void crearSesion() {
        if (usr != null)
            //Crea la sesión de usuario en el sistema
            //...
        else {
            //Excepción
        }
    }

    /**
     * Añade evento de inicio de sesión.
     */
    private void addEventoSesionInicio() {
        // Actualiza el log de eventos de sesión
        // ...

    }
}
```

La vista:

```
/**
 * MVC clase para la vista de inicio de sesión en modo texto.
 */
public class Vista {

    private static Console teclado;


    /**
     * Constructor
     */
    public Vista() {
        teclado = System.console();
```

```
    }

    /**
     * Pide el nombre corto de usuario
     * @return idCorto - tecleado por el usuario
     */
    public String pedirIdCorto() {
        String id = null;
        if (teclado != null)
            id = teclado.readLine("Teclea nombre corto de usuario:");
        return id;

    }

    /**
     * Pide la contraseña de usuario
     * @return ClaveAcceso - tecleado por el usuario
     */
    public String pedirClaveAcceso() {
        System.out.print(teclado +"\nTeclea la contraseña de usuario:");
        return new String(teclado.readPassword());
    }

    /**
     * Muestra un mensaje por consola
     * @param mensaje - A mostrar por consola
     */
    public void mostrar(String mensaje) {
        System.out.println(mensaje);
    }
}
```

Patrones de diseño estructural

Son los patrones de diseño software orientados a la solución de problemas de organización del acoplamiento entre clases y objetos; se basan en relaciones de tipo jerárquico:

- Uso
- Composición y agregación
- Herencia

Adapter o Wrapper

El patrón *Adaptador o Envoltorio*, se utiliza para transformar la interfaz de una clase preexistente (adaptada). La nueva clase (adaptadora) proporciona una interfaz que resulta diferente a la hora de su utilización según la finalidad prevista. Hay adaptaciones para distintos fines:

- Simplificación de uso o reducción del acoplamiento.
- Simple renombrado de operaciones.
- Reducción del número de operaciones o métodos.
- Ampliación de operaciones o métodos de la clase adaptada.

Esquema, participantes y colaboraciones (Adaptador de objeto)

„ Los clientes llaman a las operaciones de un objeto de la clase adaptadora. A su vez, en los métodos de la clase adaptadora se llaman a las operaciones de un objeto interno de la clase

adaptada, que tratan la petición. La relación entre la clase adaptada y adaptadora es de composición.

Ejemplo: Adaptador de objeto

```
import java.util.Calendar;

import java.util.GregorianCalendar;

// Clase adaptadora para simplificar el interfaz
// de la clase Calendar.

public class Fecha {

   // Objeto de la clase adaptada.
   private Calendar calendario;

       /**
       * Constructor defecto.
       */
       public Fecha() {
              calendario = new GregorianCalendar();
       }

       /**
        * Constructor convencional adaptado.
        */
       public Fecha(int año, int mes, int dia) {
              calendario = new GregorianCalendar(año, mes-1, dia);
       }

       //Métodos de acceso adaptados
       public int getAño() {
              return calendario.get(GregorianCalendar.YEAR);
       }

       public int getMes() {
              return calendario.get(GregorianCalendar.MONTH) + 1;
       }

       public int getDia() {
              return calendario.get(GregorianCalendar.DAY_OF_MONTH);
       }

       public void setAño(int año) {
              calendario.set(GregorianCalendar.YEAR, año);
       }

       public void setMes(int mes) {
              calendario.set(GregorianCalendar.YEAR, mes);
       }

       public void setDia(int dia) {
              calendario.set(GregorianCalendar.DAY_OF_MONTH, dia);
       }

       //.....
```

```
} // class
```

Esquema, participantes y colaboraciones (Adaptador de clase)

„ Los clientes llaman a las operaciones de un objeto de la clase adaptadora. A su vez, en los métodos de la clase adaptadora se llaman a otras operaciones heredadas de la clase adaptada, que tratan la petición. La clase adaptadora hereda de la adaptada con una relación de especialización.

Bridge

El patrón *Puente*, desacopla una abstracción de su implementación.

Composite

El patrón *Objeto Compuesto*, permite tratar objetos compuestos como si fuese uno simple.

Decorator

El patrón *Decorador*, añade dinámicamente la funcionalidad a una clase.

Façade

El patrón *Fachada*, proporciona una interfaz unificada y simplificada para acceder a una funcionalidad genérica de un subsistema; reduce la complejidad que habría que manejar si se accediera directamente al subsistema. También minimiza el acoplamiento y dependencias entre las partes.

Se aplica:

- Cuando se necesite proporcionar una interfaz simple para un subsistema complejo.
- Cuando se quiera estructurar varios subsistemas en capas, ya que las fachadas serían el punto de entrada a cada nivel.
- Otro escenario idóneo para su aplicación surge de la necesidad de desacoplar un sistema de sus clientes y de otros subsistemas, haciéndolo más independiente, portable y reutilizable (esto es, reduciendo dependencias entre los subsistemas y los clientes).

Esquema, participantes y colaboraciones

Los participantes son:

- *Fachada* (*Façade*): conoce qué clases del subsistema son responsables de una determinada función, y delega esas peticiones de los clientes a los objetos apropiados del subsistema. Actúa de intermediario.
- Las clases del subsistema que implementan la funcionalidad especializada. Realizan las funciones pedidas por la *Fachada*.

Las colaboraciones:

- Los *Clientes* se comunican con el subsistema exclusivamente enviando peticiones al objeto *Fachada* -nunca directamente-,; el objeto Fachada las reenvía a los objetos apropiados del subsistema.
- Los objetos del subsistema realizan el trabajo final, y la fachada hace algo de trabajo para acoplar su propio interfaz al del subsistema.
- Los *Clientes* que usan la *Fachada* no deben acceder directamente a los objetos del subsistema. Los clientes se abstraen de los detalles del subsistema. Ese es el objetivo del patrón: la independencia de la implementación concreta del subsistema.

Ejemplo: Fachada en la capa de acceso a datos

```
/**
 * clase Fachada para simplificar y aislar el acceso al
 * almacenamiento de datos de una aplicación.
 */
public class FachadaDatos {

	private UsuariosDAO almacenUsuarios;

	public Fachada() {
	   almacenUsuarios = UsuariosDAO.getInstancia(); // singleton
	}

	//MÉTODOS FACHADA
	//****************


	/**
	*  Da de alta un nuevo objeto en la lista.
	*  Devuelve la posición de almacenamiento.
	*  Si ya existe devuelve -1.
	*/
	public int altaUsuario(Nombre nombre) {
	   return almacenUsuarios.nuevo(nombre);
	}

	/**
	*  Guarda lista de objetos de manera persistente.
	*/
	public void guardarUsuarios() {
	   almacenUsuarios.guardarDatos();
	}

	/**
	*  Recupera datos desde el sistema de almacenamiento persistente.
	*/
	public void recuperarUsuarios() {
	   almacenUsuarios.recuperarDatos();
	}


	//...

}







/**
*  DAO (Data Access Object) para objetos Usuario
*  Almacena en un ArrayList persistente en fichero con instancia única.
*/
```

```
import java.util.ArrayList;
import java.io.FileInputStream;
import java.io.FileOutputStream;

class UsuariosDAO {

	// Requerido -patrón singleton-
	private static UsuariosDAO instancia = null;

	// Almacén de datos resuelto con ArrayList
	private ArrayList<Usuario> listaUsuarios;

	// Fichero físico para la persistencia de datos
	private String nombreFichero;
	private File fUsuarios;

	/**
	* Constructor por defecto de uso interno -patrón singleton-.
	* Sólo se ejecutará una vez.
	*/
	private UsuariosDAO() {
		listaUsuarios = new ArrayList<Usuario>();

		nombreFichero = "usuarios.dat";
		fUsuarios = new File(nombreFichero);
	}

   /**
    *  Método estático de acceso a la instancia única.
    *  Si no existe la crea invocando al constructor interno.
    *  Utiliza inicialización diferida.
    *  Sólo se crea una vez; instancia única -patrón singleton-
    *  @return instancia
    */
   public static UsuariosDAO getInstancia() {
       if (instancia == null) {
           instancia = new UsuariosDAO();
       }
       return instancia;
   }

   //MÉTODOS DAO
   //************

	/**
	 * Devuelve listado completo de usuarios.
	*/
	public String listarObjetos() {
		StringBuilder texto = new StringBuilder();
		for (int i = 0; i < listaNombres.size()
				&& listaNombres.get(i) != null; i++) {
			texto.append(listaNombres.get(i).toString() + "--");
		}
		return texto.toString();
	}
```

```
    /**
     *  Devuelve el nombre buscado.
     *  Si no existe devuelve null.
     *  Hace una búsqueda binaria.
     *  @return (UsuarioDTO)
     */
    public UsuarioDTO buscar(String nombre) {
        int comparacion;
        int inicio = 0;
        int fin = listaUsuarios.size() - 1;
        int medio;

        while (inicio <= fin) {
            medio = (inicio + fin) / 2;
            comparacion = ListaUsuarios.get(medio).getNombre().compareTo(nombre);

            if (comparacion == 0) {
                return new UsuarioDTO(listaUsuarios.get(medio));
            }

            if (comparacion < 0) {
                inicio = medio + 1;
            {
            else {
                fin = medio - 1;
            }
        }

        return null;
    }

    //Persistencia de objetos serializados en ficheros

    /**
     *  Guarda el arraylist de usuarios en fichero.
     */
    private void persistir() {

              try {
                    //Prepara fichero de salida vinculado al File fUsuarios
                    FileOutputStream fo = new FileOutputStream(fUsuarios);

                    // Configura el stream de objetos
                    ObjectOutputStream osUsuarios = new ObjectOutputStream(fo);

                    //Serializa el ArrayList al fichero
                    osUsuarios.writeObject(listaUsuarios);

                    osUsuarios.flush();
                    osUsuarios.close();
              }
              catch (IOException e) {
                    e.printStackTrace();
              }
    }

    /**
```

```
    *  Recupera el Arraylist de nombres almacenados en fichero.
    */
       public void recuperar() {

              //Limpiar el ArrayList de posible basura
              listaUsuarios.clear();

              try {
                     if (fUsuarios.exists()) {
                           //Prepara fichero de entrada
                           FileInputStream fi = new FileInputStream(fUsuarios);

                           //Configura el stream de objetos
                           ObjectInputStream isUsuarios = new
                           ObjectInputStream(fi);

                           //Regenera el ArrayList desde el fichero
                           listaNombres =
                                  (ArrayList<Usuarios>) isUsuarios.readObject();

                           isUsuarios.close();
                     }
                     else
                           throw new FileNotFoundException();
              }
              catch (ClassNotFoundException e) {
                     e.printStackTrace();
              }
              catch (FileNotFoundException e) {
                     e.printStackTrace();
              }
              catch (IOException e) {
                     e.printStackTrace();
              }
       }

} // class


/**
 *  Clase del modelo.
 */
class Usuario implements Serializable {

       private String nombre;

       public Usuario() {
              this.nombre = " ";
       }

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

      public void setNombre(String nombre) {
              this.nombre = nombre;
       }
```

```
} // class


class UsuarioDTO {

	private String nombre;

	public UsuarioDTO(Usuario usuario) {
		this.nombre = usuario.getNombre();
	}

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

	public void setNombre(String nombre) {
		this.nombre = nombre;
	}


} // class
```

Flyweight

El patrón *Peso Ligero*, reduce la redundancia cuando gran cantidad de objetos poseen idéntica información.

Proxy

El patrón *Representante*, mantiene a un representante de un objeto.

Módulo

El patrón *Módulo*, agrupa varios elementos relacionados, como clases, singletons, y métodos, utilizados globalmente, en una entidad única.

Patrones de comportamiento

Se definen como patrones de diseño software que ofrecen soluciones respecto a la interacción, responsabilidades y cohesión entre clases y objetos, así como los algoritmos que encapsulan.

Chain of Responsibility

El patrón *Cadena de Responsabilidad*, permite establecer la línea que deben llevar los mensajes para que los objetos realicen la tarea indicada.

Command

El patrón *Orden*, encapsula una operación en un objeto, permitiendo ejecutar dicha operación sin necesidad de conocer el contenido de la misma.

Interpreter

El patrón *Intérprete*, define una gramática para un lenguaje dado, así como las herramientas necesarias para interpretarlo.

Iterator

El patrón *Iterador*, permite realizar recorridos sobre objetos compuestos independientemente de la implementación de estos.

Mediator

El patrón *Mediador*, define un objeto que coordine la comunicación entre objetos de distintas clases, pero que funcionan como un conjunto.

Memento

El patrón *Recuerdo*, permite volver a estados anteriores del sistema.

Observer

El patrón *Observador*, define una dependencia de uno-a-muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y actualicen automáticamente todos los objetos que dependen de él.

State

El patrón *Estado*, permite que un objeto modifique su comportamiento cada vez que cambie su estado interno.

Strategy

El patrón *Estrategia*, permite disponer de varios métodos para resolver un problema y elegir cuál utilizar en tiempo de ejecución.

Template Method

El patrón *Método Plantilla*, define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos, esto permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
Se usa en los siguientes casos:

- Para implementar algoritmos parecidos en clases diferentes. Las partes del algoritmo que no cambian se colocan en una clase común y retrasar la implementación completa en las subclases.
- Cuando un comportamiento se repite en varias subclases debería factorizarse y ser localizado en una clase común para evitar código repetido.
- Controlar la extensión de subclases definiendo una plantilla de método que llama a operaciones vacías en puntos específicos permitiendo extensiones solo en estos puntos.

Visitor

El patrón *Visitante*, permite definir nuevas operaciones sobre una jerarquía de clases sin modificar las clases sobre las que opera.

Patrones de interacción

El primer intento por aplicar este concepto en el diseño de las interfaces de usuario se dio por Ward Cummingham y Kent Beck quienes adaptaron la propuesta de C. Alexander y crearon cinco patrones de interfaz: Window per task, Few panes, Standard panes, Nouns and verbs, y Short Menu. En años más recientes investigadores como Martin Van Welie, Jennifer Tidwell han desarrollado colecciones de patrones de interacción para la World Wide Web. En dichas colecciones captan la experiencia de programadores y diseñadores expertos en el desarrollo de interfaces usables y condensan esta experiencia en una serie de guías o recomendaciones, que puedan ser usadas por los desarrolladores novatos con el propósito de que en poco tiempo adquieran la habilidad de diseñar interfaces que inciden en la satisfacción de los usuarios. Los patrones de interacción buscan la reutilización de interfaces eficaces y un manejo óptimo de los recursos de las páginas web, haciendo más eficaz el consumo de tiempo en el diseño del sitio web y permitiendo a los programadores novatos adquirir más experiencia.

Ejercicios

1. (***) Busca información del patrón *Abstract Factory* y aplicarlo en las clases de los ejemplos.
    - ❖ Se puede leer sobre el patrón *Abstract Factory* en Wikipedia: http://en.wikipedia.org/wiki/Abstract_factory_pattern

Fuentes y bibliografía

- NAKOV, S. *Fundamentos de programación con Java*. Ed. Faber, Veliko Tarnovo, 2009. [en linea][traducción automática del búlgaro] *http://www.introprogramming.info/intro-java-book/*
- SZNAJDLEDER, P.A. Java a fondo. Alfaomega y Marcombo S.A. 2ª ed. 2013. ISBN: 978-84-267-1976-2
- [VIDAL1] VIDAL GARCÍA, A. *Codecriticon. Introducción a los Patrones de Diseño*. [Blog] http://codecriticon.com/introduccion-patrones-diseno/
- PAVÓN, J. *Patrones de diseño orientado a objetos*
  Patrón Modelo Vista Controlador

- DAMIÁN CAMPOS, D. *Patrones de diseño, refactorización y antipatrones.*
- Wikipedia. *Patrones de diseño.* [en línea] http://es.wikipedia.org/wiki/Categor%C3%ADa:Patrones_de_dise%C3%B1o
- KUMAR, P. *Design Patterns*. [en línea] http://www.journaldev.com/dev/java/design-patterns
- GAMMA, E. HELM, R. JOHNSON, R. y VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*, Ed. Addison-Wesley Iberoamericana, ISBN 84-7829-059-1
- Oracle. *Java™ Platform, Standard Edition 8* API Specification. [en línea] http://docs.oracle.com/javase/8/docs/api/