Introducción a los patrones de diseño para problemas orientados a objetos

[3.1] ¿Cómo estudiar este tema?

[3.2] Qué son los patrones de diseño

[3.3] Patrón `Factory`

[3.4] Patrón `Singleton`

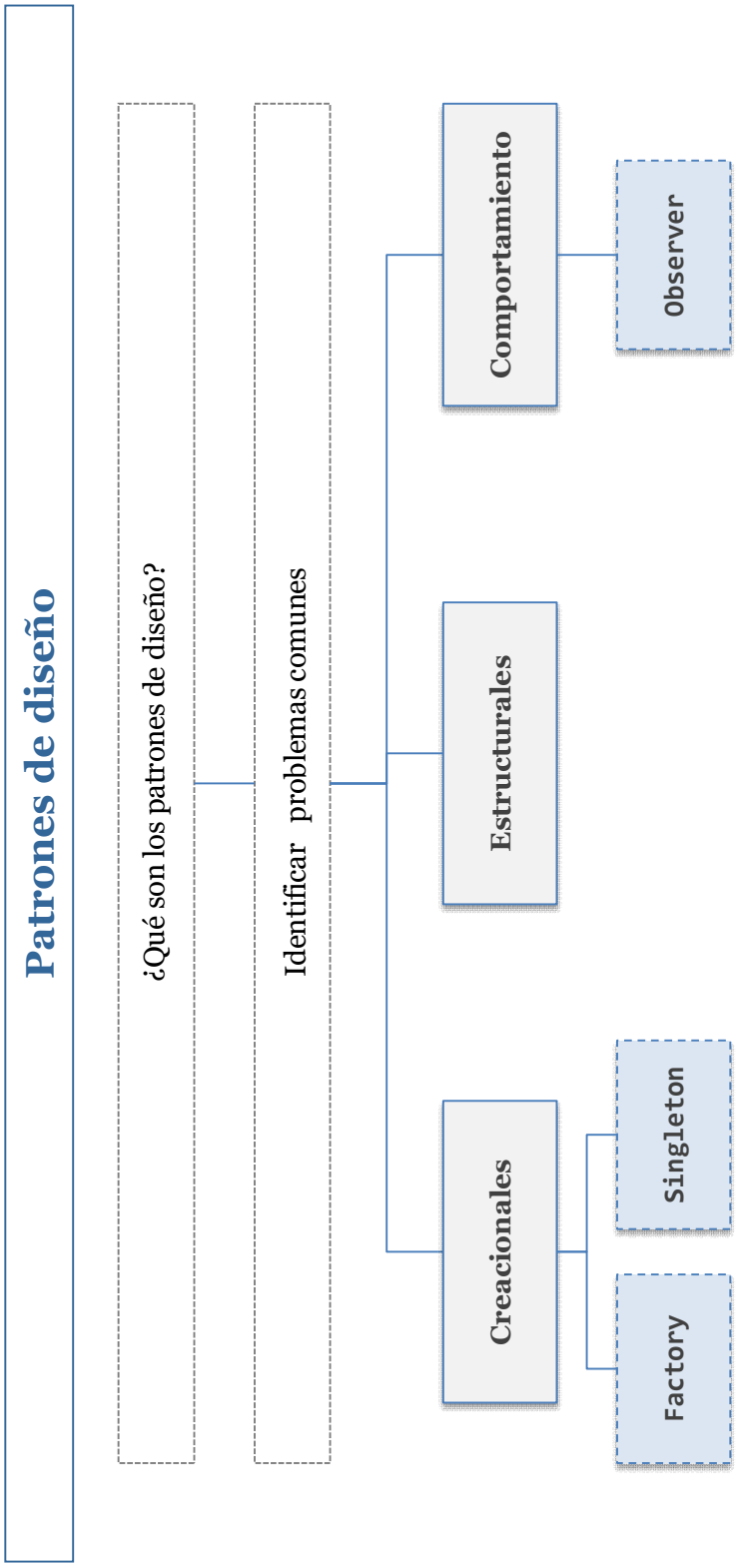
[3.5] Patrón `Observer`

[3.6] Patrón `Composite`

[3.7] Introducción a la POO distribuida

TEMA 3

Esquema



Ideas clave

3.1. ¿Cómo estudiar este tema?

Para estudiar este tema debes leer los puntos dedicados a encapsulamiento, herencia, clases abstractas, polimorfismo, herencia simple y patrones de diseño **(páginas 388-394)** del siguiente libro, disponible en el aula virtual bajo licencia CEDRO:

Pressman, R. S. (2002). *Ingeniería del software: un enfoque práctico*. Madrid: McGraw Hill.

También debes leer el capítulo 8, «Patrones y principios de diseño» **(páginas 141-151)** del siguiente libro, disponible en la Biblioteca Virtual de UNIR:

Vélez, J. (2011). *Diseñar y programar, todo es empezar: Una introducción a la programación orientada a objetos usando UML y Java*. Madrid: Dykinson.

Para comprobar si has comprendido los conceptos puedes realizar el test de autoevaluación del tema.

En este tema vamos a implementar en Java algunos de los principales patrones de diseño:

- Reutilización de código a través de patrones de diseño.
- Implementación de patrones de diseño para un problema en concreto.

3.2. Qué son los patrones de diseño

Los **patrones de diseño** son soluciones reutilizables que se encuentran en los problemas habitualmente.

Los patrones de diseño fueron propuestos por el Gang of Four (GoF), en su libro *Design Patterns. Elements of Reusable Object-Oriented Software*. El término GoF hace referencia a los cuatro autores del libro: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

Los patrones se componen de:

- **Nombre**: identifica el problema y la solución. Suele usarse el nombre original en inglés del libro de GoF.
- **Problema**: describe la situación y las condiciones en las que es conveniente utilizar el patrón.
- **Solución**: describe los elementos del diseño de la solución, relaciones, responsabilidades y colaboraciones. La solución no describe una implementación.
- **Consecuencias**: describen las ventajas e inconvenientes de aplicar el patrón.

Existen muchos tipos de patrones de diseño:

- **Patrones estructurales:** Separan la interfaz de la implementación. Nos aseguran independencia entre las capas *software* que se van creando.
- **Patrones creacionales**: Inicialización y configuración de objetos.
- **Patrones de comportamiento**: Más que describir objetos o clases, describen la comunicación entre ellos.

3.3. Patrón `factory`

El patrón `factory` o `factory Method` es un patrón creacional.

El patrón define una interfaz para la creación de objetos, permitiendo que sean las subclase las que decidan qué clase necesitan instanciar.

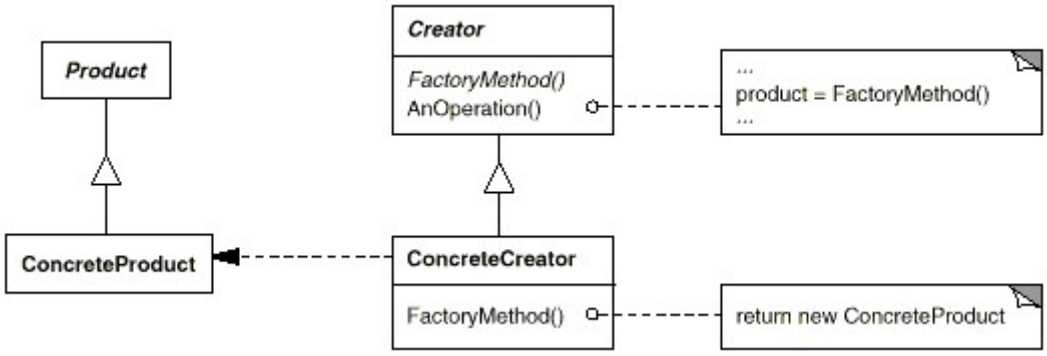


Diagrama OMT de `Factory Method`. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). Design Patterns. Elements of Reusable Object-Oriented Software. Indiana: Addison Wesley.

Aplicabilidad

Usar cuando:

- Una clase no puede prever la clase de objetos que debe crear.
- Una clase quiere que sean sus subclases quienes especifiquen los objetos que esta crea.
- Las clases delegan la responsabilidad en una de entre varias clases auxiliares y queremos localizar concretamente en qué subclase de auxiliar se delega.

Consecuencias

- Proporciona enganches para las subclases.
- Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente.
- Conecta jerarquías de clases paralelas.

Consulta la web de Microsoft para conocer más sobre este patrón:
http://msdn.microsoft.com/es-es/library/bb972258.aspx#M21

Implementación

Su implementación es:

Ejemplo extraído de:
http://es.wikipedia.org/wiki/Factory_Method_%28patr%C3%B3n_de_dise%C3%B1o%29

```
abstract class Creator{
	// Definimos método abstracto
	public abstract Product factoryMethod();
}

public class ConcreteCreator extends Creator{
	public ConcreteProduct factoryMethod() {
		return new ConcreteProduct();
	}
}

public interface Product{
	public void operacion();
}
```



```
public class ConcreteProduct implements Product{
	public void operacion(){
		System.out.println("Una operación de este producto");
	}
}

public class Factory{
	public static void main(String args[]){
		Creator aCreator;
		aCreator = new ConcreteCreator();
		Product producto = aCreator.factoryMethod();
		producto.operacion();
	}
}
```

3.4. Patrón singleton

El patrón singleton es un **patrón creacional**. Su objetivo es **asegurar que una clase tiene una única instancia** y proporcionar un punto de acceso global a la misma.

A veces es importante asegurar que **una clase solo tiene una instancia** (por ejemplo una sola cola de impresión, un gestor de ventanas, un sistema de ficheros...).

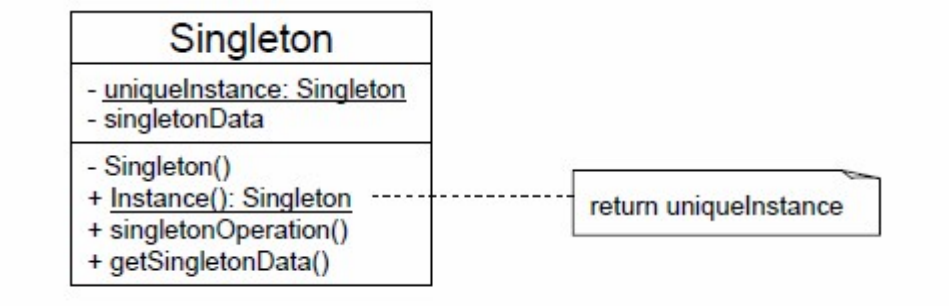


Diagrama patrón Singleton. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). Design Patterns. Elements of Reusable Object-Oriented Software. Indiana: Addison Wesley.

Aplicabilidad

Usar cuando:

- Debe haber exactamente una instancia de una clase, que debe ser accesible a los clientes a través de un punto de acceso conocido.
- La instancia única pueda tener subclases y los clientes deban ser capaces de usar las subclases sin modificar su propio código.

Consecuencias

- Controla el acceso a la instancia de la clase.
- Permite la creación de subclases.
- El acceso a la instancia de la clase se realiza mediante un método de clase o método estático.

Implementación

```
Public class Singleton {

	private static Singleton singleton = new Singleton( );

	private Singleton(){ }
//private hace que el constructor no sea invocable.


//a través del método getInstance accedemos a la única instancia del
objeto.
	public static Singleton getInstance( ) {
		return singleton;
	}


//el objeto Singleton puede tener otros métodos
	protected  void demoMethod( ) {
		System.out.println("demoMethod for singleton");
	}
}


public class SingletonDemo {
	public static void main(String[] args) {
		Singleton tmp = Singleton.getInstance( );
		tmp.demoMethod( );
	}
}
```

El patrón `singleton` lo retomaremos en el tema de programación concurrente, ya que existen algunas consideraciones a tener en cuenta cuando es accedido por hilos. Para asegurar que el objeto sea único, faltaría una única cosa. ¿Qué pasaría si clonamos el objeto?

Respuesta: Si clonamos el objeto se crearía una nueva instancia del mismo, por lo que dejaría de ser único. La solución sería sobrescribir el método clone de tal manera que no se puedan clonar.

3.5. Patrón observer

El patrón observer es un **patrón de comportamiento**.

El patrón define una **dependencia de uno-a-muchos entre objetos** (*observado-observadores*) de forma que, cuando el objeto *observado* cambia, se notifica a los objetos *observadores*.

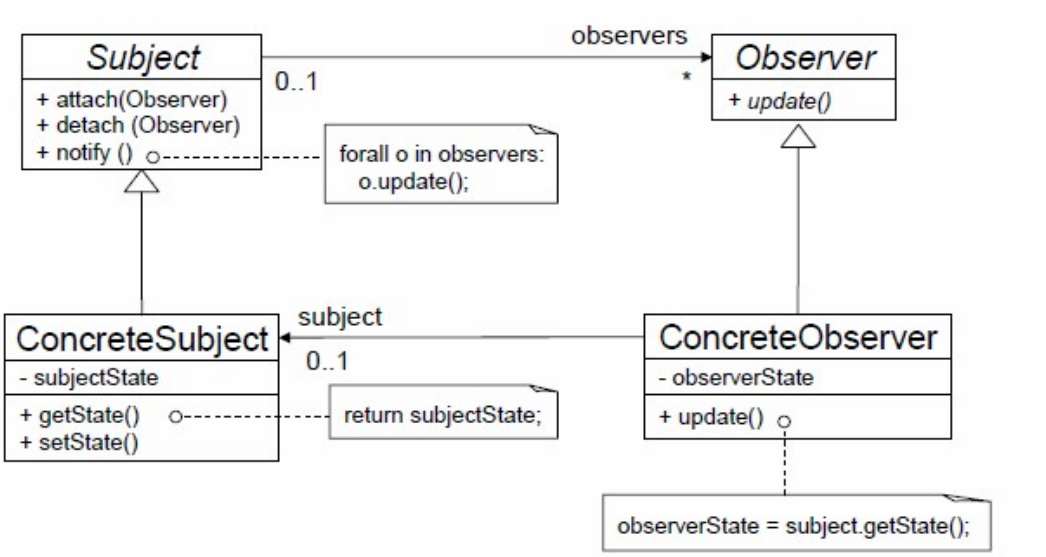


Diagrama patrón observer. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). Design Patterns. Elements of Reusable Object-Oriented Software. Indiana: Addison Wesley.

Aplicabilidad

Usar cuando:

- Los cambios de estado en un objeto (observado) requieren cambios en otros objetos (observadores).
- No se conoce el número de objetos que deben cambiar cuando cambia el objeto observado.
- Un objeto sea capaz de notificar algo a otros sin saber quiénes son esos objetos.
- Una abstracción tiene dos aspectos, que dependen uno del otro.

Consecuencias

- Permite añadir nuevos observadores sin necesidad de realizar modificaciones en el sujeto u otros observadores.
- Permite modificar sujetos y observadores de manera independiente.
- Permite bajo acoplamiento, ya que el sujeto no conoce la clase concreta de sus observadores.
- Las notificaciones de un sujeto se envía a todos sus observadores, sin necesidad de especificar los receptores de la notificación.

Implementación

En Java se han incluido librerías para dar soporte a este patrón a través de la clase `java.util.Observable` y la `interface java.util.Observer`

Ejemplo extraído del libro: Learn Java de O'reilly:
http://examples.oreilly.com/learnjava2/CD-ROM/examples/ch10/MessageBoard.java

```
import java.util.Observable;
import java.util.Observer;

class Foro extends Observable { //el objeto que queremos observar
	private String msg;

	public String getMessage() {
		return msg;
	}

	public void ActualizaForo(String msg) {
		this.msg = msg;
		//cambiamos el estado
		setChanged();
		 //notificamos a quien nos esté observando
		notifyObservers(msg);
	}

}

class Detector implements Observer { //quien lo va a observer
	public void update(Observable o, Object arg) {
//sobreescribimos el método, es la acción que se generará cuando se
detecte un cambio

		System.out.println("Nuevo mensaje: " + arg);
	}
}
```



```
public class PatronObserver{

	public static void main(String[] args) {
		Foro foro = new Foro();
		Detector d1 = new Detector();
		Detector d2 = new Detector();
		foro.addObserver(d1);
		foro.addObserver(d2);
		foro.ActualizaForo("Examen el viernes!!!");
	}
}
```

3.6. Patrón composite

El patrón composite es un patrón estructural.

Este patrón permite construir objetos compuestos en estructuras en forma de árbol, permitiendo tratar de la misma manera a los objetos simples y compuestos.

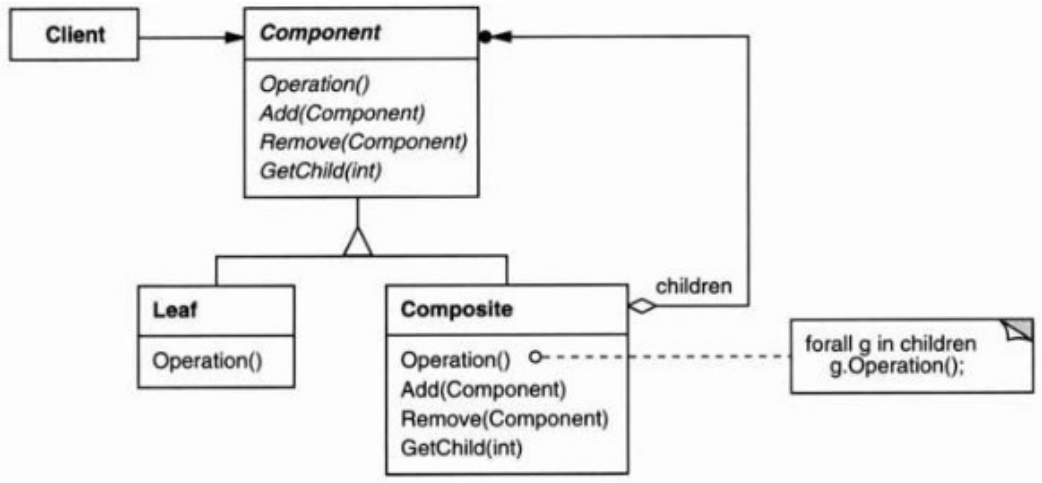


Diagrama patrón composite. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). Design Patterns. Elements of Reusable Object-Oriented Software. Indiana: Addison Wesley.

Aplicabilidad

Usar cuando:

- Se desee representar jerarquías de objetos todo-parte.
- Los objetos simples y compuestos deban ser manejados de manera uniforme.
- Se desee que los clientes puedan ignorar las diferencias entre los objetos simples y compuestos.

Consecuencias

- Define jerarquías de clases formadas por objetos simples y compuestos. Permite que en cualquier parte del código donde se espera un objeto simple, también se pueda recibir un objeto compuesto.
- Simplifica el cliente: los clientes pueden tratar uniformemente los objetos simples y compuestos.
- Facilita incorporar nuevos tipos de componentes.
- La desventaja de este patrón es que puede hacer un diseño demasiado general

Implementación

```
//interfaz Componente
public interface Componente
{
        public void getNombre();
        public void agregar(Componente c);
        public void eliminar(Componente c);
        public Componente getComponente();
}


//Componente1
public class Componente1 implements Componente{
        public Componente getComponente (){return null;}
        public void getNombre{
                System.out.println(“Componente1”);
        }
        public void agregar(Componente c){}
        public void eliminar(){}
}


//Componente2
public class Componente2 implements Componente{
        public Componente getComponente (){return null;}
        public void getNombre{
                System.out.println(“Componente2”);
        }
        public void agregar(Componente c){}

        public void eliminar(Componente c){}
}
```

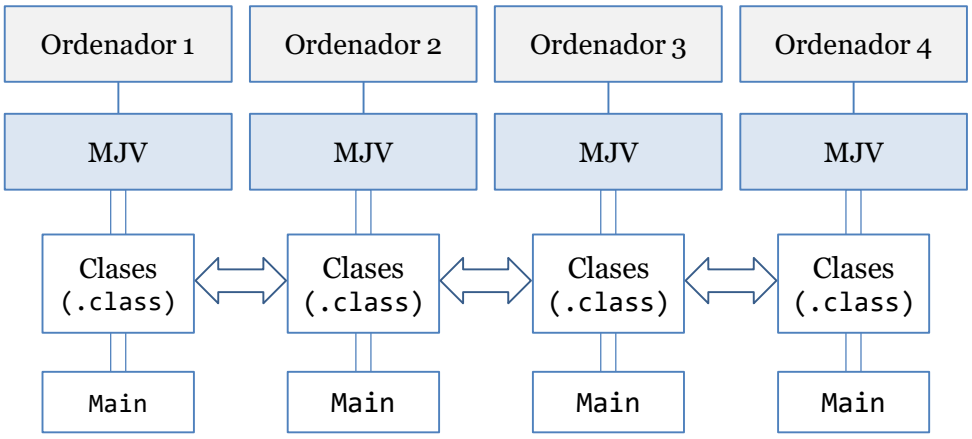


```
//Composite
import java.util.ArrayList;

public class Composite implements Componente{
        private ArrayList listaComponente= new ArrayList();
        public Composite (){
        }
        public Composite (Componente c){
                this.agregar(c);
        }

        public Componente getComponente(){
                return null;
        }
        public getNombre(){
        for (int i=0; i<listaComp.size();i++){
                ((Component)listaComponente.get(i)).getNombre();
        }
        }
        public agregar (Component c){
                listaComponente.add(c);
        }
        public eliminar (Component c){
                listaComponente.remove(c);
        }

}

//Cliente
public class Cliente {
  public static void main(String[]args){
        Componente [] c={new Componente1(), new Componente2()};
        Componente c2=new Composite();
        c2.agregar(c[0]);
        c2.agregar(c[1]);
        Componente c3=new Composite(c2);
        c3.agregar(c[0]);
        c3.getNombre();
        c3.agregar(c[1]);
        c3.eliminar(c2);
        System.out.println();
        c3.getNombre();
  }
}
```


3.7. Introducción a la POO distribuida

El concepto de **Programación Orientada a Objetos Distribuida (POOD)** en Java está basado en que varios ordenadores remotos usen la Máquina Virtual de Java (MVJ) para que puedan invocar métodos de Objetos de otras MVJ.

La siguiente figura muestra la arquitectura basada en MVJ.



Esta arquitectura funciona por medio del concepto **objeto servidor-cliente**, donde el objeto servidor invoca al objeto cliente. En este caso el objeto cliente será un ordenador remoto.

Cuando se realizan interfaces remotas, los objetos cliente nunca realizan referencia directa a otro cliente. Solo el servidor será el que tenga la configuración para que el cliente pueda acceder. Los clientes remotos se suelen llamar **Stubs,** que contienen información del host, puerto e ID del objeto. De esta manera al tener un Stub, se puede acceder a ellos.

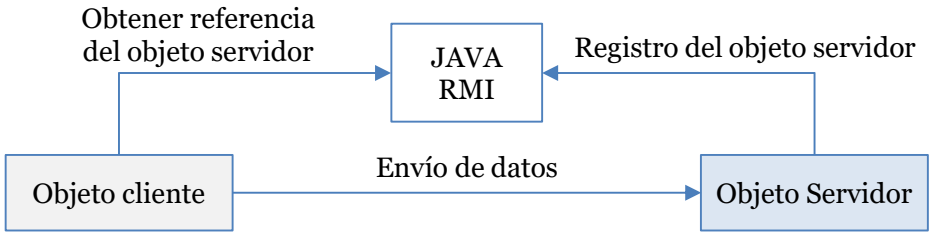
Para poder construir una aplicación POOD se puede hacer uso de varias formas de comunicación. Entre las más importantes se encuentran:

- Aplicaciones basadas en TCP/IP usando sockets y definiendo quien es el servidor y el cliente.
- Basado en CORBA (del inglés, Common Object Request Broker Architecture). CORBA está diseñada para admitir un modelo de objetos de multiplataforma con lo cual permite que los dos sistemas sean eficaces.

- Basado en el RMI de Java es una de las alternativas más eficientes para el procesamiento de información distribuida.

La arquitectura de JAVA RMI permite tener capas independientes teniendo definido el tipo de protocolo e interfaz de tal manera que una capa modificada no afectara a las otras.

Por ejemplo, si una capa tiene un protocolo definido en UDP no afectara a otra que esté definida en TCP/IP. Esto es de vital importancia, ya que hay varias arquitecturas de otros sistemas que no permiten tener capas de comunicación con diferentes protocolos. En la siguiente figura muestra un ejemplo de como un objeto servidor-cliente se comunican a través de Java RMI.



Lo + recomendado

No dejes de leer…

Importancia de los patrones de diseño

O'Reilly, T. (marzo 2007). What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software. *International Journal of Digital Economics* 65, 17-37.

Paper en el que se habla de la importancia de los patrones de diseño en la nueva generación del *software*.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://mpra.ub.uni-muenchen.de/4578/

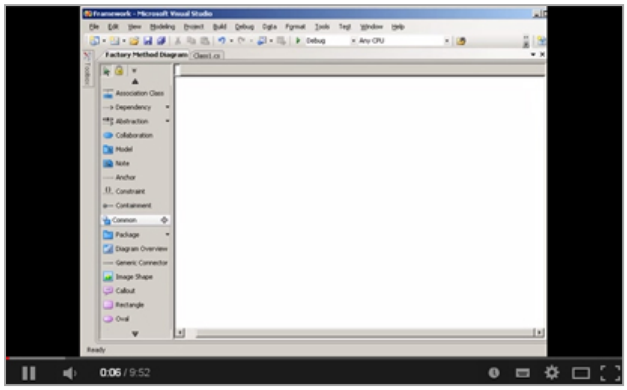
Patrones

Consulta la página de Microsoft en la que podrás encontrar información sobre los patrones y sus tipos.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://msdn.microsoft.com/es-es/library/bb972240.aspx

No dejes de ver...

Patrón `Factory`



Vídeo en el que se explica el patrón de diseño `Factory`.

Accede al vídeo desde el aula virtual o a través de la siguiente dirección web:
https://www.youtube.com/watch?v=7KFBiR4gZzU

+ Información

A fondo

Patrones de fabricación

Página de Microsoft en la que podrás encontrar información detallada sobre los patrones de fabricación.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://msdn.microsoft.com/es-mx/library/bb972258.aspx

Patrón `Singleton`

Página de Microsoft en la que podrás encontrar información detallada sobre el patrón `singleton`.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://msdn.microsoft.com/es-es/library/bb972272.aspx

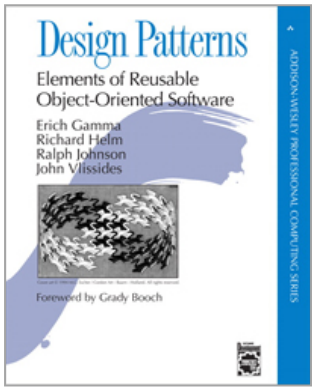
Patrón `Observer`

Página de Microsoft en la que podrás encontrar información detallada sobre el patrón `observer`.

Accede al artículo desde el aula virtual o a través de la siguiente dirección web:
http://msdn.microsoft.com/es-es/library/bb972192.aspx

Diseño de patrones

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Indiana: Addison Wesley.

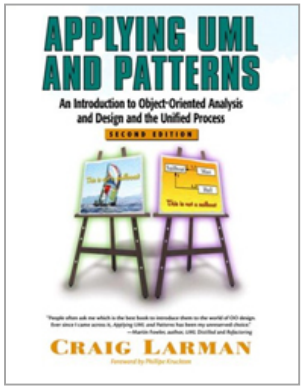


Manual sobre ingeniería del *software* que describe soluciones a problemas habituales utilizando patrones de diseño.

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web:
http://books.google.es/books?id=6oHuKQe3TjQC&printsec=frontcover

Implementar patrones en UML

Larman, C. (2001). *Applying UML and patterns*. Nueva Jersey: Prentice Hall.

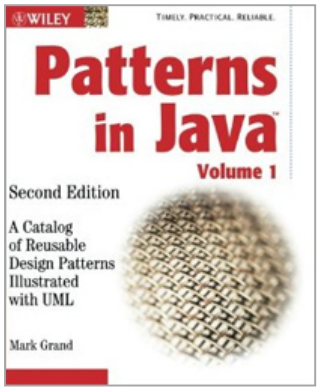


Libro que en el que se hace una explicación muy extensa sobre los patrones y cómo implementarlos en UML

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web:
http://books.google.es/books?id=r8i-4En_aa4C&printsec=frontcover

Patrones en Java

Grand, M. (2002). *Patterns in Java: A catalog of reusable design patterns illustrated with UML*. Indiana: Wiley & Sons.



Libro en el que se hace una exposición de los patrones en Java, con ejemplos prácticos para su reutilización.

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web:
http://books.google.es/books?id=810chlr-IQgC&printsec=frontcover

Patrones de diseño en Java

Debrawer , L. (2013). *Patrones de diseño en Java*. Barcelona: Editorial ENI.

Este libro presenta de forma concisa y práctica los veintitrés modelos de diseño (*Design Patterns*) fundamentales ilustrándolos mediante ejemplos adaptados y fáciles de comprender. Cada ejemplo se describe en UML y en Java bajo la forma de un pequeño programa completo y ejecutable.

Accede a una parte del libro desde el aula virtual o a través de la siguiente dirección web:
http://books.google.es/books?id=x8wRTkfnU4kC&printsec=frontcover

Recursos externos

Eclipse

Eclipse es una plataforma de desarrollo. Fue concebida desde sus orígenes para convertirse en una plataforma de integración de herramientas de desarrollo. No tiene en mente un lenguaje específico, sino que es un IDE genérico.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:
https://www.eclipse.org/downloads/

En el siguiente vídeo puedes ver cómo crear un proyecto en Eclipse desde cero:
https://www.youtube.com/watch?v=J9lkAK0L16I

Test

1. Los patrones de diseño se utilizan para:

A. Se utilizan para solucionar problemas comunes en el desarrollo del *software*.

B. Se utilizan para la toma de requisitos de los problemas.

C. Se utilizan para la toma de decisiones a la hora la definición de clases.

D. Se utilizan para ahorrar espacio en los programas.

2. Los tipos de patrones son:

A. Estructurales, creacionales, de comportamiento y de estado.

B. Estructurales, creacionales y de comportamiento.

C. Estructurales, de estado y creacionales.

D. Creacionales, de estado y de comportamiento.

3. El patrón `factory method` es de tipo:

A. Creacional.

B. Estructural.

C. Estado.

D. Comportamiento.

4. El patrón `factory method` se utiliza para:

A. Asegurar que una clase tiene una única instancia.

B. Crear objetos dentro de una clase que no sabe qué tipo de objetos debe crear.

C. Asegurar el comportamiento concurrente de un objeto.

D. Asegurar la respuesta en tiempo real de un objeto.

5. El patrón `singleton` es de tipo:

A. Creacional.

B. Estructural.

C. Estado.

D. Comportamiento.

6. El patrón `singleton` se usa para:

A. Asegurar que una clase tiene una única instancia.

B. Construir objetos de manera personalizada.

C. Asegurar que un objeto se comunica solo con otro objeto.

D. Asegurar que un objeto se crea a través de su constructor.

7. El patrón `observer` es de tipo:

A. Creacional.

B. Estructural.

C. Estado.

D. Comportamiento.

8. El patrón `observer` se utiliza para:

A. Se utiliza para crear objetos únicos.

B. Se utiliza para comprobar si un objeto cambia de estado.

C. Se utiliza para crear objetos de manera concurrente.

D. Se utiliza para comprobar que un objeto se ha creado correctamente.

9. Los patrones estructurales se utilizan para:

A. Separan la interfaz de la implementación.

B. Hacen que la interfaz y la implementación estén unidas.

C. Permiten crear estructuras nuevas en los programas.

D. No existen.

10. Los patrones de comportamiento:

A. Definen cómo se deben crear los objetos y clases.

B. Son reglas de documentación de programas.

C. Son reglas de nomenclatura de variables y constantes.

D. Definen las reglas de comunicación de los elementos implicados en un problema.

