Diseño de clases avanzadas

Diseño de clases avanzadas

Modelación de problemas de negocio con clases

La herencia (o creación de subclases) es una función esencial del lenguaje de programación Java. La herencia permite la reutilización de código mediante:

- Herencia de métodos: las subclases evitan la duplicación del código al heredar las implantaciones de métodos.
- Generalización: el código que está diseñado para basarse en el tipo más genérico posible es más fácil de mantener.

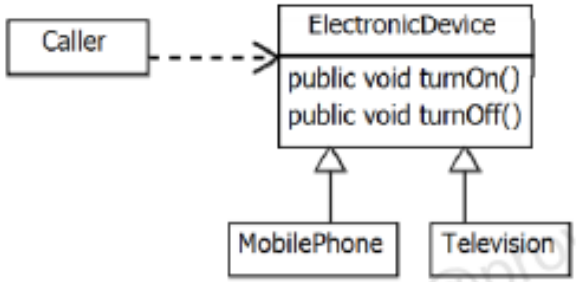

Diagrama de herencia de clases

Diseño de clases avanzadas

Activación de la generalización

La codificación en un tipo base común permite introducir nuevas subclases con pocas modificaciones o ninguna de cualquier código que dependa del tipo base más genérico.

```
ElectronicDevice dev = new Television();
dev.turnOn(); // all ElectronicDevices can be turned on
```

Use siempre el tipo de referencia más genérico posible.

Diseño de clases avanzadas

Definición de clases abstractas

Una clase se puede declarar como abstracta mediante el modificador de nivel de clase `abstract`.

```
public abstract class ElectronicDevice { }
```

- Una clase abstracta puede tener una subclase.

```
public class Television extends ElectronicDevice { }
```

- Una clase abstracta no se puede instanciar.

```
ElectronicDevice dev = new ElectronicDevice(); // error
```

Diseño de clases avanzadas

Definición de métodos abstractos

Un método se puede declarar como abstracto mediante el modificador de nivel de método `abstract`.

```
public abstract class ElectronicDevice {

   public abstract void turnOn();
   public abstract void turnOff();
}
```

Sin corchetes

Un método abstracto:

- No puede tener un cuerpo de método.
- Se debe declarar en una clase abstracta.
- Se sobrescribe en subclases.

Diseño de clases avanzadas

Validación de clases abstractas

Las siguientes reglas adicionales se aplican cuando se usan las clases y los métodos abstractos:

- Una clase abstracta puede tener cualquier número de métodos abstractos y no abstractos.
- Al heredar de una clase abstracta, debe realizar una de las siguientes acciones:
  - Declarar la clase secundaria como abstracta.
  - Sustituya todos los métodos abstractos heredados de la clase principal. De lo contrario, se producirá un error de tiempo de compilación.

```
error: Television is not abstract and does not override
abstract method turnOn() in ElectronicDevice
```

Diseño de clases avanzadas

Palabra clave `static`

El modificador `static` se usa para declarar campos y métodos como recursos de nivel de clase. Los miembros de clase estáticos:

- Se pueden usar sin instancias de objeto.
- Se usan cuando un problema se soluciona mejor sin objetos.
- Se usan cuando hay objetos del mismo tipo que deben compartir campos.
- *No* se deben usar para no usar las funciones orientadas a objetos de Java a menos que haya un motivo justificado.

Diseño de clases avanzadas

Métodos estáticos

Los métodos estáticos son métodos que se pueden llamar incluso si la clase en la que se hayan declarado no se ha instanciado. Los métodos estáticos:

- Se denominan métodos de clase.
- Son útiles para las API que no están orientadas a objetos.
  - `java.lang.Math` contiene muchos métodos estáticos.
- Se suelen usar en lugar de los constructores para realizar tares relacionadas con la inicialización de objetos.
- No pueden acceder a miembros no estáticos de la misma clase.

Diseño de clases avanzadas

Implantación de métodos estáticos

```
public class StaticErrorClass {
    private int x;

    public static void staticMethod() {
        x = 1; // compile error
        instanceMethod(); // compile error
    }

    public void instanceMethod() {
        x = 2;
    }
}
```

Diseño de clases avanzadas

Llamada a métodos estáticos

```
double d = Math.random();
StaticUtilityClass.printMessage();
StaticUtilityClass uc = new StaticUtilityClass();
uc.printMessage(); // works but misleading
sameClassMethod();
```

Al llamar a los métodos estáticos, debería:

- Cualificar la ubicación del método con un nombre de clase si el método se encuentra en otra clase distinta a la del emisor de la llamada
  - No es necesario para métodos de la misma clase
- Evitar el uso de una referencia de objeto para llamar a un método estático

Diseño de clases avanzadas

Variables estáticas

Las variables estáticas son variables a las que se puede acceder incluso aunque la clase en la que se hayan declarado no se haya instanciado. Las variables estáticas:

- Se denominan variables de clase.
- Se limitan a una sola copia por JVM.
- Son útiles para contener datos compartidos.
  - Los métodos estáticos almacenan datos en variables estáticas.
  - Todas las instancias de objetos comparten una sola copia de cualquier variable estática.
- Se inicializan cuando la clase contenedora se carga por primera vez.

Diseño de clases avanzadas

Definición de variables estáticas

```
public class StaticCounter {
    private static int counter = 0;

    public StaticCounter() {
        counter++;
    }

    public static int getCount() {
        return counter;
    }
}
```

Solo una copia en la memoria

Diseño de clases avanzadas

Uso de variables estáticas

```
double p = Math.PI;
```

```
new StaticCounter();
new StaticCounter();
System.out.println("count: " + StaticCounter.getCount());
```

Al acceder a las variables estáticas, debería:

- Cualificar la ubicación de la variable con un nombre de clase si la variable se encuentra en otra clase distinta a la del emisor de la llamada
  - No es necesario para variables de la misma clase
- Evitar el uso de una referencia de objeto para acceder a una variable estática

Diseño de clases avanzadas

Importaciones estáticas

Una sentencia de importación estática hace que los miembros estáticos de una clase estén disponibles con su nombre simple.

- Con cualquiera de las siguientes líneas:

```
import static java.lang.Math.random;
import static java.lang.Math.*;
```

- La llamada al método `Math.random()` se podría escribir como:

```
public class StaticImport {
    public static void main(String[] args) {
        double d = random();
    }
}
```

Diseño de clases avanzadas

Métodos finales

Un método se puede declarar como `final`. Los métodos finales no se pueden sobrescribir.

```
public class MethodParentClass {
    public final void printMessage() {
        System.out.println("This is a final method");
    }
}
```

```
public class MethodChildClass extends MethodParentClass {
    // compile-time error
    public void printMessage() {
        System.out.println("Cannot override method");
    }
}
```

Diseño de clases avanzadas

Clases finales

Una clase se puede declarar como `final`. Las clases finales no se pueden ampliar.

```
public final class FinalParentClass { }
```

```
// compile-time error
public class ChildClass extends FinalParentClass { }
```

Diseño de clases avanzadas

Variables finales

El modificador `final` se puede aplicar a las variables. Las variables finales no pueden cambiar sus valores una vez inicializadas. Las variables finales pueden ser:

- Campos de clase
  - Los campos finales con expresiones de constantes de tiempo de compilación son variables de constantes.
  - Los campos estáticos se pueden combinar con los finales para crear una variable siempre disponible y que nunca cambia.
- Parámetros del método
- Variables locales

Diseño de clases avanzadas

Declaración de variables finales

```
public class VariableExampleClass {
    private final int field;
    private final int forgottenField;
    private final Date date = new Date();
    public static final int JAVA_CONSTANT = 10;

    public VariableExampleClass() {
        field = 100;
    }

    public void changeValues(final int param) {
        param = 1; // compile-time error
        date.setTime(0); // allowed
        date = new Date(); // compile-time error
        final int localVar;
        localVar = 42;
        localVar = 43; // compile-time error
    }
}
```

Diseño de clases avanzadas

Patrones de diseño

Los patrones de diseño:

- Son soluciones reutilizables a problemas de desarrollo de software comunes.
- Están documentados en catálogos de patrones.
  - *Design Patterns: Elements of Reusable Object-Oriented Software* (Patrones de diseño: elementos del software reutilizable orientado a objetos), de Erich Gamma et al. (conocido como "Gang of Four", la banda de los cuatro, por sus cuatro autores)
- Forman un vocabulario para hablar sobre el diseño.

Patrones de diseño

- Hay disponibles catálogos de patrones para muchos lenguajes de programación. La mayoría de los patrones de diseño tradicionales se aplican a cualquier lenguaje de programación orientado a objetos. Una de las obras más conocidas, Design Patterns: Elements of Reusable Object-Oriented Software, usa una combinación de C++, Smalltalk y diagramas para mostrar las posibles implementaciones de patrones. Son muchos los desarrolladores Java que aún hacen referencia a esta obra, ya que los conceptos se pueden extrapolar a cualquier lenguaje orientado a objetos.

Patrones de diseño

- Los patrones para el desarrollo de software son uno de los últimos avances de la Tecnología Orientada a Objetos. Los patrones son una forma literaria para resolver problemas de ingeniería del software, que tienen sus raíces en los patrones de la arquitectura.

- Los diseñadores y analistas de software más experimentados aplican de forma intuitiva algunos criterios que solucionan los problemas de manera elegante y efectiva. La ingeniería del software se enfrenta a problemas variados que hay que identificar para poder utilizar la misma solución (aunque matizada) con problemas similares.

Patrones de diseño

- Las metodologías Orientadas a Objetos tienen como uno de sus principios “no reinventar la rueda” para la resolución de diferentes problemas. Por lo tanto los patrones se convierten en una parte muy importante en las Tecnologías Orientadas a Objetos para poder conseguir la reutilización.

- El objetivo de los patrones es crear un lenguaje común a una comunidad de desarrolladores para comunicar experiencia sobre los problemas y sus soluciones.

Patrones de diseño - Definiciones

- Los diferentes autores han dado diversas definiciones de lo que es un patrón software. Veamos a continuación alguna de ellas:
- 1. Dirk Riehle y Heinz Zullighoven:
- Un patrón es la abstracción de una forma concreta que puede repetirse en contextos específicos.
- 2. Otras definiciones más aceptadas por la comunidad software son:
- Un patrón es una información que captura la estructura esencial y la perspicacia de una familia de soluciones probadas con éxito para un problema repetitivo que surge en un cierto contexto y sistema.
- Un patrón es una unidad de información nombrada, instructiva e intuitiva que captura la esencia de una familia exitosa de soluciones probadas a un problema recurrente dentro de un cierto contexto.

Patrones de diseño - Definiciones

- 3. Según Richard Gabriel:
- Cada patrón es una regla de tres partes, la cual expresa una relación entre un cierto contexto, un conjunto de fuerzas que ocurren repetidamente en ese contexto y una cierta configuración software que permite a estas fuerzas resolverse por si mismas.
  Esta definición es similar a la dada por Alexander: Cada patrón es una relación entre un cierto contexto, un problema y una solución. El patrón es, resumiendo, al mismo tiempo una cosa que tiene su lugar en el mundo, y la regla que nos dice cómo crear esa cosa y cuándo debemos crearla. Es al mismo tiempo una cosa y un proceso; al mismo tiempo una descripción que tiene vida y una descripción del proceso que la generó.

Patrones de diseño

- Una cosa que los diseñadores expertos no hacen es resolver cada problema desde el principio. A menudo reutilizan las soluciones que ellos han obtenido en el pasado. Cuando encuentran una buena solución, utilizan esta solución una y otra vez. Consecuentemente tu podrías encontrar patrones de clases y comunicaciones entre objetos en muchos sistemas orientados a objetos.

Patrones de diseño

- Estos patrones solucionan problemas específicos del diseño y hacen los diseños orientados a objetos más flexibles, elegantes y por último reutilizables. Los patrones de diseño ayudan a los diseñadores a reutilizar con éxito diseños para obtener nuevos diseños.

  Un diseñador que conoce algunos patrones puede aplicarlos inmediatamente a problemas de diseño sin tener que descubrirlos.

Patrones de diseño

- El objetivo de los patrones de diseño es guardar la experiencia en diseños de programas orientados a objetos. Cada patrón de diseño nombra, explica y evalúa un importante diseño en los sistemas orientados a objetos.

- Es decir se trata de agrupar la experiencia en diseño de una forma que la gente pueda utilizarlos con efectividad. Por eso se han documentado los mas importantes patrones de diseño y presentado en catálogos.

Patrones de diseño - Singleton

- Objetivo : Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él. Todos los objetos que utilizan una instancia de esa clase usan la misma instancia.

- Aplicabilidad : El patrón Singleton se puede utilizar en los siguientes casos:

✓ Debe haber exactamente una instancia de la clase.

✓ La única instancia de la clase debe ser accesible para todos los clientes de esa clase.

Patrones de diseño - Singleton

- Solución: El patrón Singleton es relativamente simple, ya que sólo involucra una clase.

Una clase Singleton tiene una variable static que se refiere a la única instancia de la clase que quieres usar. Esta instancia es creada cuando la clase es cargada en memoria.

Se debe implementar la clase de tal forma que se prevenga que otras clases puedan crear instancias adicionales de una clase Singleton. Esto significa que debes asegurarte de que todos los constructores de la clase son privados.

Para acceder a la única instancia de una clase Singleton, la clase proporciona un método static, normalmente llamado getInstancia, el cual retorna una referencia a la única instancia de la clase

Patrones de diseño - Singleton

- Consecuencias :

1. Existe exactamente una instancia de una clase Singleton.
2. Otras clases que quieran una referencia a la única instancia de la clase Singleton conseguirán esa instancia llamando al método static getInstancia de la clase. Controla el acceso a la única instancia.
3. Tener subclases de una clase Singleton es complicado y resultan clases imperfectamente encapsuladas. Para hacer subclases de una clase Singleton, deberías tener un constructor que no sea privado. También, si quieres definir una subclase de una clase Singleton que sea también Singleton, querrás que la subclase sobrescriba el método getInstancia de la clase Singleton. Esto no será posible, ya que métodos como getInstancia deben ser static. Java no permite sobrescribir los métodos static.

Singleton - Implementación

El patrón de diseño singleton detalla una implantación de clase que solo se puede instanciar una vez.

```
public class SingletonClass {
1   private static final SingletonClass instance =
            new SingletonClass();

2   private SingletonClass() {}

3   public static SingletonClass getInstance() {
        return instance;
    }
}
```

Singleton - Implementación

1. Utilice una referencia estática para apuntar a la instancia única. Declarar la referencia como final garantiza que nunca se hará referencia a otra instancia.

2. Agregue un solo constructor privado a la clase singleton. El modificador privado solo permite acceso de la "misma clase", lo que prohíbe cualquier intento de instanciar la clase singleton, excepto en el caso del intento en el paso 1.

Singleton - Implementación

3. Un método de fábrica público devuelve una copia de la referencia a singleton. Este método se declara como estático para acceder al campo estático declarado en el paso 1. El paso 1 podría usar una variable pública, lo que hace que no sea necesario el método de fábrica.

Los métodos de fábrica ofrecen una mayor flexibilidad (por ejemplo, implantar una solución singleton por thread) y se suelen usar en la mayoría de las implantaciones de singleton.

Para obtener una referencia de singleton, llame al método getInstance :

```
SingletonClass ref = SingletonClass.getInstance();
```