Módulo 2

Principios y patrones de diseño en OOP

Tecnara – Cursos de formación

Índice de contenidos

integra

tecnara
Cluster TIC de Aragón / Aragón IT Cluster

01 Introducción

integra

Introducción

El paradigma de Programación Orientada a Objetos

- Comenzó a finales de la década de 1950
- Se hizo popular en la década de 1970
- Dominó el desarrollo de programas desde la década de 1990

En todo ese tiempo, hay autores que han reconocido errores comunes y problemas comunes, a los que han dado solución mediante buenas prácticas y patrones de diseño, respectivamente

Introducción

Se supone que un programador novel (denominado Junior) que aprende OOP va a escribir programas que:

- funcionen PERFECTAMENTE para resolver el problema planteado en el enunciado…
- … pero cualquier modificación o adición de funcionalidad que haya que realizar en el futuro obligará a hacer varios cambios en distintos puntos del programa

Introducción

Se supone que conforme el programador Junior adquiere experiencia (deja de ser Junior) va a darse cuenta del problema anterior y va a modificar su código para no tener que hacer tantos cambios cuando toque

Con el objetivo de ayudar a que los Junior dejen de ser considerados como tal en el menor tiempo posible, varios expertos dedicados a la informática han escrito:

- Manuales de buenas prácticas, conocidos como **principios de diseño**
- Recetas para solucionar problemas habituales, conocidas como **patrones de diseño**

Introducción

Sobre principios y patrones hay muchísima bibliografía, cursos y recursos online

En este tema veremos, bastante por encima, tres (y medio):

- Principios SOLID
- Principios GRASP

- Patrones de diseño del Gang of Four

- DDD (Domain-Driven Design, o Diseño Dirigido por el Dominio)

Introducción

La idea de este tema no es convertiros de repente en buenos programadores, porque eso viene con el tiempo, con la práctica y cometiendo muchos errores

Se pretende dar un vistazo a conceptos y problemas que os podéis encontrar en algún momento, para que cuando los necesitéis os suenen de algo, sepáis que alguien ya se molestó en plantear una solución, y podáis buscar y adaptar esa solución para vuestro problema

Insisto, la mayoría de lo que se va a ver en este tema es probable que no lo vayáis a aplicar en un programa nunca, o durante mucho tiempo, y aún así el programa funcionará bien.

Introducción

El uso de principios y patrones de diseño OOP da al código:

- Mayor modularidad: una separación más clara entre las distintas partes
- Mayor reusabilidad: la separación anterior facilita llamar a métodos desde varios sitios
- Mayor escalabilidad: menos cambios necesarios si hay que ampliar funcionalidad
- Mayor mantenibilidad: menos cambios necesarios si hay que cambiar funcionalidad
- Mayor testabilidad: menos código a escribir para pruebas unitarias y de integración

02

Principios SOLID

Principios SOLID

Robert C. Martin, conocido en el mundo de la informática como ‘el tío Bob’ (Uncle Bob) es un ingeniero de software norteamericano, famoso por dos artículos:

- Design Principle and Design Patterns (2000), con 5 principios para diseño OOP
- The Agile Manifesto (2001), con 12 principios para definir una metodología ágil

Principios SOLID

El primer artículo contenía los 5 principios del diseño de programación orientada a objetos que han acabado por llamarse principios SOLID.

La palabra es un acrónimo formado por la 1º letra de cada uno de los cinco principios:

- **S**ingle-responsibility principle
- **O**pen-closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Principios SOLID: S

Single-responsibility principle

Cada clase debe ser responsable de una única parte de la funcionalidad del programa, y todos los atributos y métodos que contenga deben estar relacionados con esa única responsabilidad

Principios SOLID: S

Single-responsibility principle

- Un ejemplo muy común: separar en clases diferentes la parte de hacer cálculos u operaciones que generen un resultado, y la parte de pedir información o mostrar el resultado por pantalla.

```
public class OperacionesPersonas {
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
}
```

```
public class InfoPorPantallaPersonas {
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: S

Single-responsibility principle

- Esto permite que, si en el futuro no queremos pedir información por pantalla o mostrar el resultado por pantalla, sino que lo queremos leer y escribir en ficheros o en base de datos, podamos crear una nueva clase para obtener la información, y no cambiar nada en la clase que hace las operaciones

```
public class OperacionesPersonas {
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
}
```

```
public class InfoEnFicheroPersonas {
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: S

El planteamiento es el siguiente:

- Analizamos una clase cualquiera
- Para cada método se anotan los distintos recursos que utiliza en su código:
  - Parámetros de entrada, o recursos iniciales
  - Recursos que usa para obtener el resultado, aparte de los parámetros
  - Parámetro de salida, o recurso que utiliza para transmitir el resultado

```
public class GestionPersonas {
    public List<Persona> ListaP { get; set; }
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: S

El planteamiento es el siguiente:

Parámetros de entrada (recursos iniciales)
Recursos adicionales usados para obtener el resultado
Recurso usado para transmitir el resultado

Persona
Lista Personas
Lista Personas

Persona, Lista Personas

-
Pantalla, Persona
Persona

Pantalla, Persona

```
public class GestionPersonas {
    public List<Persona> ListaP { get; set; }
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: S

El planteamiento es el siguiente:

Parámetros de entrada (recursos iniciales)
Recursos adicionales usados para obtener el resultado
Recurso usado para transmitir el resultado

Persona
Pantalla
Pantalla

} Persona, Pantalla

-
Lista Personas
Pantalla

} Lista Personas, Pantalla

```
public class GestionPersonas {
    public List<Persona> ListaP { get; set; }
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: S

Si queremos separar los métodos que usan la pantalla, porque en un futuro igual cambia la salida a pantalla por salida a fichero, el análisis anterior localiza los métodos que la usan, y gracias a ello podemos sacarlos a otra clase diferente

En este caso, la 'Single Responsibility' sería la salida por pantalla

```
public class GestionPersonas {
    public List<Persona> ListaP { get; set; }
    public void AnnadirPersona(Persona p){…}
    public void ModificarPersona(Persona p) {…}
    public void BorrarPersona(Persona p) {…}
    public Persona ObtenerInfoPersona() {…}
    public void MostrarInfoPersona(Persona p) {…}
    public void MostrarListaPersonas() {…}
}
```

Principios SOLID: O

Open-closed principle (no planteado por Robert Martin, sino por Bertrand Meyer)

- Cada clase debe permitir que otras clases la extiendan (herencia)
- Cada método debe permitir que otros métodos lo extiendan (sobreescritura de método en clases hijas)

- Hay que evitar, en lo posible, modificar la propia clase o el propio método. Para eso, las clases padre abstractas o las interfaces OBLIGAN a que uno o varios métodos no puedan cambiar su declaración (sí permiten cambiar su implementación, pero si el programador cumple este principio, en vez de modificar la implementación original creará otra clase hija con un método que sobreescribirá esa implementación)

Este principio lo cumpliremos siempre que hagamos uso de herencia y polimorfismo

Principios SOLID: O

Open-closed principle: sobre la extensión

- Supongamos que tenemos una clase que hasta el momento ha servido para lo que hemos necesitado

```
public class Persona {
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
    public bool EsMayorDeEdad() {…}
}
```

Principios SOLID: O

Open-closed principle: sobre la extensión

- De repente, esa clase no es suficiente, y queremos añadirle algún atributo o algún método

```
public class Persona {
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
    public decimal CocienteIntelectual { get; set; }
    public bool EsMayorDeEdad() {…}
    public bool EsSuperdotada() {…}
}
```

Principios SOLID: O

Open-closed principle: sobre la extensión

- El principio Open-closed nos PROHIBE tocar ni una coma de la clase original.

```
public class Persona {
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
    public decimal CocienteIntelectual { get; set; }
    public bool EsMayorDeEdad() {…}
    public bool EsSuperdotada() {…}
}
```

Principios SOLID: O

Open-closed principle: sobre la extensión

- En su lugar, nos OBLIGA a crear una clase heredada que añada o modifique lo necesario

```
public class Persona {
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
    public bool EsMayorDeEdad() {…}
}

public class PersonaConIQ: Persona {
public decimal CocienteIntelectual { get; set; }
public bool EsSuperdotada() {…}
}
```

Principios SOLID: O

Open-closed principle: sobre la modificación

- Para evitar que algún programador caiga en la tentación de cambiar la signatura:
  - Nombre
  - parámetros de entrada
  - parámetro de salida

  en un método de una clase ya existente, este principio aconseja el uso de clases abstractas o interfaces para que cualquier cambio en la signatura de un método genere un error de compilación que obligue a dejarlo como estaba

```
public interface IPersona {
    bool EsMayorDeEdad();
}

public class Persona {
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public int Edad { get; set; }
    public bool EsMayorDeEdad() {…}
}
```

Principios SOLID: L

Liskov substitution principle (no planteado por Robert Martin, sino por Barbara Liskov)

Tiene una formulación matemática muy llamativa (no por ello atractiva):

- Sea **φ(x)** una propiedad comprobable acerca de los objetos **x** de tipo **T**. Entonces **φ(y)** debe ser verdad para los objetos **y** del tipo **S** donde **S** es un subtipo de **T**

Vamos a desentrañar esta maraña matemática, a ver qué narices nos está contando…

Para ello, nada mejor que un caso CONCRETO de aplicación (sin letras x, y, T, S, φ)

Principios SOLID: L

Liskov substitution principle

- Sea **φ(x)** una propiedad comprobable acerca de los objetos **x** de tipo **T**. Entonces **φ(y)** debe ser verdad para los objetos **y** del tipo **S** donde **S** es un subtipo de **T**

```
public abstract class PersonaConBeneficios {
   private decimal Ingreso;

   public void AddIngreso(decimal nuevoIngreso)
   {   Ingreso += nuevoIngreso;   }

   public abstract decimal GetBeneficio();
}
```

```
public class PersonaMenorDeEdad {
   public override decimal GetBeneficio()
   {   return Ingreso; // la paga semanal acumulada   }
```

```
public class PersonaAdulta {
   public override decimal GetBeneficio()
   {   return Ingreso-Impuestos-CosteDeVida;   }
```

```
public class PersonaJubilada {
   public override decimal GetBeneficio()
   {   return Ingreso-CosteDeVida;   }
```

Principios SOLID: L

Liskov substitution principle

- Sea **φ(x)** una propiedad comprobable acerca de los objetos **x** de tipo **T**. Entonces **φ(y)** debe ser verdad para los objetos **y** del tipo **S** donde **S** es un subtipo de **T**

- T: clase PersonaConBeneficios
- x: un objeto de clase PersonaConBeneficios
- S: varias subclases de PersonaConBeneficios (PersonaMenorDeEdad, PersonaAdulta, PersonaJubilada)
- y: un objeto de alguna de las subclases (PersonaMenorDeEdad, o PersonaAdulta, o PersonaJubilada)
- φ: que el resultado devuelto por el método GetBeneficio() definido en PersonaConBeneficios devuelva siempre un valor positivo o cero (nunca negativo)

Principios SOLID: L

Liskov substitution principle

Reescribimos el principio con los valores concretos:

- Supongamos que GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase PersonaConBeneficios. Entonces GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase PersonaMenorDeEdad, o PersonaAdulta, o PersonaJubilada

- ¿Esto se cumple siempre? Vamos subclase por subclase…
  - Si el coste de vida (comida, ropa, facturas, hipoteca, etc) más los impuestos superan los ingresos, el valor devuelto por GetBeneficio() es negativo…

Principios SOLID: L

Liskov substitution principle

Reescribimos el principio con los valores concretos:

- Supongamos que GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase PersonaConBeneficios. Entonces GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase **PersonaMenorDeEdad**

- ¿Esto se cumple siempre?
  - Sí, se cumple siempre. Un menor de edad poco espabilado podrá tener ingresos (con paga semanal) o no, y decidir un mes gastarse todos sus ahorros en un capricho, pero nunca se podrá comprar algo que exceda sus ahorros, y nadie le obliga a pagar más de lo que tiene

Principios SOLID: L

Liskov substitution principle

Reescribimos el principio con los valores concretos:

- Supongamos que GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase PersonaConBeneficios. Entonces GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase **PersonaAdulta**

- ¿Esto se cumple siempre?
  - NO, no se cumple siempre. Si el coste de vida (comida, ropa, facturas, hipoteca, etc) más los impuestos superan los ingresos (salario si trabaja, prestación por desempleo si está desempleado y ha solicitado un subsidio, o NADA si no trabaja y no ha solicitado un subsidio), el valor devuelto por GetBeneficio() es negativo

Principios SOLID: L

Liskov substitution principle

Reescribimos el principio con los valores concretos:

- Supongamos que GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase PersonaConBeneficios. Entonces GetBeneficio() nunca debe dar resultado negativo para ningún objeto de clase **PersonaJubilada**

- ¿Esto se cumple siempre?
  - NO, no se cumple siempre. Si el coste de vida (comida, ropa, facturas, hipoteca, etc) supera los ingresos (pensión si ha trabajado, ayudas del Estado si las ha solicitado, o NADA si solo vive de sus ahorros), el valor devuelto por GetBeneficio() es negativo

Principios SOLID: L

Liskov substitution principle (no planteado por Robert Martin, sino por Barbara Liskov)

El análisis anterior nos llevaría a no admitir como subclases posibles de PersonaConBeneficios a las clases PersonaAdulta y PersonaJubilada, ya que al menos existe una propiedad de obligado cumplimiento según la definición de la clase padre, que no es seguro que se vaya a cumplir SIEMPRE para objetos de alguna de las dos clases hijas mencionadas

Principios SOLID: L

Así pues, este principio nos da una herramienta para comprobar si, al añadir una nueva clase hija a una clase abstracta que ya tenía otras hijas anteriormente, la implementación de un método sobreescrito por esta nueva clase va a ser 'compatible' con las otras implementaciones hechas por las otras clases hijas, de forma que no pueda dar ningún error si se aplica polimorfismo para llamar indistintamente al método sobreescrito por cualquiera de las clases hijas, tanto la nueva como las antiguas

```
public class GestionBanco {
   public void Ejecutar() {
      List<PersonasConBeneficio> lpb = new List<PersonasConBeneficio>() {
      new PersonaMenorDeEdad(), new PersonaAdulta(), new PersonaJubilada()
      };
      lpb.ForEach( x => Banco.Ingresar(x.GetBeneficio()) ); // si Banco.Ingresar() no admite negativos, el programa puede dar error
   }
}
```

Principios SOLID: L

El ejemplo anterior plantea una fuente de error basada en postcondición (resultado devuelto), pero el principio considera cuatro posibles fuentes de error:

- Todos los métodos de las clases hijas deben tener una precondición más débil que el método correspondiente de la clase padre.
- Todos los métodos de las clases hijas deben tener una postcondición más fuerte que el método correspondiente de la clase padre
- Las invariantes establecidas en la clase padre deben ser mantenidas por las clases hijas
- Restricción (o regla) histórica: las clases hijas no pueden hacer cambios en atributos definidos en la clase padre si la clase padre define que no se permiten cambios en dichos atributos

Principios SOLID: L

Todos los métodos de las clases hijas deben tener una **precondición más débil** que el método correspondiente de la clase padre. Ejemplo:

- Si el método de la clase padre admite como entrada enteros no negativos, no puede haber ningún hijo que para ese método dé error si recibe como entrada un número no negativo

Si método padre admite entrada de rango [N,M], entonces métodos hijos deben admitir entrada de rango igual o mayor

	OK	KO
Entrada en método padre	[0,inf)	[0,inf)
Entrada en métodos hijos	(-inf,inf)	[1,inf)

Principios SOLID: L

Todos los métodos de las clases hijas deben tener una **postcondición más fuerte** que el método correspondiente de la clase padre. Ejemplo:

- Si el método de la clase padre da como resultado solo enteros no negativos, no puede haber ningún hijo que para ese método pueda generar como salida correcta un número negativo

Si método padre admite salida de rango [N,M], entonces métodos hijos deben admitir salida de rango igual o menor

	OK	KO
Salida en método padre	[0,inf)	[0,inf)
Salida en métodos hijos	[1,inf)	(-inf,inf)

Principios SOLID: L

Las **invariantes** establecidas en la clase padre deben ser mantenidas por las clases hijas. Ejemplo:

- Si la clase padre solo tiene sentido si una de sus propiedades es siempre no negativa, no puede haber ningún método en las clases hijas que pueda asignar a esa propiedad un valor negativo

	OK	KO
Rango posible para una propiedad	[0,inf)	[0,inf)
Valor cambiado por hijo	1	-1

Principios SOLID: L

Restricción (o regla) histórica: las clases hijas no pueden hacer cambios en los atributos de la clase padre que la clase padre no defina como cambios permitidos. Ejemplo:

- Si la clase padre tiene un atributo que debe tener siempre el mismo valor, no puede haber ninguna clase hija con algún método que cambie el valor de ese atributo.

Principios SOLID: I

Interface segregation principle

Mejor muchas interfaces con pocos métodos definidos, que pocas interfaces con muchos métodos definidos

- La idea es que cuando una clase implementa una interfaz está obligada a implementar todos sus métodos. Si una interfaz define muchos métodos, es posible que queramos crear una clase que vaya a heredar de esa interfaz, pero no nos interese implementar todos los métodos definidos en ella… En ese caso, mejor separar esos métodos en dos o más interfaces, y que nuestra clase implemente solo las interfaces que definan los métodos que nos interesan.

Principios SOLID: I

Interface segregation principle

Ejemplo:

```
public interface ICalculosGeometricos {
    decimal Area();
    decimal Volumen();
}
```

```
public class Cuadrado: ICalculosGeometricos {
    public decimal lado { get; set; }
    public decimal Area() { return lado*lado; }
    public decimal Volumen() { ¿¿…?? }
}
```

```
public class Cubo: ICalculosGeometricos {
    public decimal lado { get; set; }
    public decimal Area() { return lado*lado; }
    public decimal Volumen() { return lado*lado*lado; }
}
```

- Clases Cuadrado y Cubo. ¿Tiene sentido que Cuadrado implemente Volumen()? NO

Principios SOLID: D

Dependency inversion principle

Los parámetros de los métodos definidos en una clase deberían ser de un tipo ‘interfaz’, de una clase abstracta o de una clase heredada cuando desde la invocación de ese método el parámetro correspondiente pueda ser de clases distintas.

NOTA: cuando los parámetros de tipo interfaz o clase abstracta se encuentran en un constructor, significa que al crear objetos de la clase, la invocación al constructor debe especificar qué clase concreta de las posibles que implementen la interfaz o hereden la clase abstracta se quiere añadir como propiedad del objeto que se esté creando. Es como si desde la invocación se estuviese ‘inyectando’ la clase deseada para ese atributo. Por ello, la inversión de dependencias, aplicada a un constructor de clase, se denomina **inyección de dependencias (Dependency Injection, o DI)**

Principios SOLID: D

Ejemplo de inversión de dependencia:

```
public class GestionBanco {
   public void Ingresar(PersonaMenorDeEdad nueva)
   {   Banco.Add(cliente.GetBeneficio());   }
   public void Ingresar(PersonaAdulta nueva)
   {   Banco.Add(cliente.GetBeneficio());   }
   public void Ingresar(PersonaJubilada nueva)
   {   Banco.Add(cliente.GetBeneficio());   }

   public void Ingresar(PersonaConBeneficios cliente)
   {   Banco.Add(cliente.GetBeneficio());   }
}
```

Rojo: Sin inversión de dependencia (clases hijas)
Verde: Con inversión de dependencia (clase padre abstracta o interfaz)

```
public abstract class PersonaConBeneficios {
   private decimal Ingreso;

   public void AddIngreso(decimal nuevoIngreso)
   {   Ingreso += nuevoIngreso;   }

   public abstract decimal GetBeneficio();
}
```

```
public class PersonaMenorDeEdad {
   public override decimal GetBeneficio()
   {   return Ingreso; // la paga semanal acumulada   }
```

```
public class PersonaAdulta {
   public override decimal GetBeneficio()
   {   return Ingreso-Impuestos-CosteDeVida;   }
```

```
public class PersonaJubilada {
   public override decimal GetBeneficio()
   {   return Ingreso-CosteDeVida;   }
```

03

Principios GRASP

Principios GRASP

General **R**esponsibility **A**ssignment **S**oftware **P**atterns, de Craig Larman

Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development (2005)

Principios GRASP

- A diferencia de los principios SOLID, que aportan criterios generales para el diseño de clases reusables y ampliables sin que esto genere errores indeseados, los principios GRASP son algo más práctico, más concreto.

- Tratan de dar respuesta a preguntas específicas que nos podemos plantear en el diseño de esas clases

- GRASP da respuesta a 9 de esas preguntas

Principios GRASP

GRASP da respuesta a 9 de esas preguntas, cada una asignada a uno de sus 9 principios:

- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

Principios GRASP: Controller

Pregunta: ¿Qué clase debe encargarse de gestionar los datos de entrada que se obtienen en la capa de presentación (introducidos por pantalla, leídos de fichero, …)?

- Para abstraer la capa de presentación de la capa de negocio (donde se realizan los cálculos que necesita cada operación de nuestro programa), es conveniente añadir clases que actúen como intermediarias entre el formato de envío de los datos desde la capa de presentación y el formato en el que esperan recibir los datos los métodos de la capa de negocio. Estas clases intermedias se llaman Controladores.

- Los controladores cobran especial importancia en el desarrollo web con el empleo de la arquitectura MVC (Modelo, Vista, Controlador)

Principios GRASP: Creator

Pregunta: ¿Qué clase debe encargarse de crear nuevos objetos (nuevas instancias) de alguna otra clase?

- Cuando hay varias clases entre las que se duda cuál debería poder crear objetos de otra clase, la respuesta que da Creator es: En general, la clase A que debe encargarse de crear objetos de clase B es la que cumpla más de las siguientes condiciones:
  - Alguna de las propiedades de A es una lista o similar de elementos de tipo B
  - Alguna de las propiedades de A es de tipo B
  - Algún método de A contiene la información necesaria para crear un elemento de tipo B
  - Algún método de A guarda elementos de tipo B
  - Algún método de A utiliza en algún momento elementos de tipo B

Principios GRASP: Indirection

Pregunta: ¿Qué clase debe encargarse de transmitir datos enviados por una interfaz a otra interfaz que espera los datos en un formato diferente?

- Es una generalización del principio Controller (que se aplicaba a la comunicación Presentación-Negocio)
- Igual que se plantea crear una clase intermedia para comunicar estas dos capas, también se plantea crear una capa intermedia para comunicar Negocio-Datos, o Negocio-Dominio y Dominio-Datos, o en general, allí donde se perciba una diferencia entre la interfaz* origen y la interfaz* destino de una transferencia de datos

* NOTA: en este caso el concepto interfaz se refiere a la estructura que tienen los datos que se envían desde una capa, y la estructura que tienen que tener en la capa a la que tienen que llegar

Principios GRASP: Information expert

Pregunta: ¿Qué clase debe encargarse de definir como propiedades los datos y definir como métodos las operaciones que plantee el enunciado del problema?

- Es una generalización del principio Creator (que se aplicaba a la elección de la clase que debe incluir en alguno de sus métodos la creación de objetos de otra clase)

- La clase más adecuada para añadir como propiedad otra clase o añadir como método una operación que trabaje con esa otra clase, será la clase que más información posea sobre la otra clase con la que va a trabajar, o dicho de otro modo, la experta en información sobre la otra clase. Se comprueban las mismas condiciones que en Creator

Principios GRASP: High cohesion

Pregunta: ¿Cómo conseguir clases modulares, dedicadas a una funcionalidad común o al manejo de un único recurso (la Single Responsibility de los principios SOLID)?

- Añadir a cada clase métodos que tengan una alta cohesión entre sí. Evitar clases que puedan hacer cosas muy diferentes. No mezclar churras con merinas.

- Una clase con alta cohesión tiene métodos que permiten realizar operaciones accediendo o modificando un conjunto de propiedades igual o similar

- Una clase con baja cohesión tiene métodos que no tienen relación unos con otros, o que incluye propiedades a las que nunca se accede desde distintos métodos de la misma

Principios GRASP: Low coupling

Pregunta: ¿Cómo conseguir clases con baja dependencia de otras clases, que no obliguen a cambiar mucho código si cambia alguna de las clases de las que depende (lo que facilitará mucho la reusabilidad de la clase en el futuro)?

- Evitar siempre que sea posible que una clase tenga atributos de otra clase, o que sus métodos tengan parámetros o devuelvan resultados de otra clase. Mejor que los atributos, los parámetros o los resultados sean de alguna clase abstracta, o mejor de alguna interfaz.

- Esto permitirá poder cambiar de una clase a otra que implemente la misma interfaz o sea hija de la misma clase abstracta, y no habrá errores ni necesidad de cambios

Principios GRASP: Polymorphism

Pregunta: ¿Cómo asegurar que varias clases que implementan métodos con funcionalidad equivalente pueden ser invocados sin tener que conocer a priori cuál se va a ejecutar?

- Básicamente, sobre lo que ya conocemos sobre polimorfismo en la Programación Orientada a Objetos, este principio no se limita a decirnos que existe la posibilidad de usarlo si nos apetece, sino que nos anima a utilizarlo siempre que tenga sentido hacerlo, para simplificarnos la vida (nos la complicamos ahora para simplificárnosla si en un futuro toca realizar cambios).

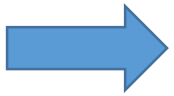
Principios GRASP: Protected variations

Pregunta: ¿Cómo asegurar que al crear una nueva clase, cambios futuros en esta clase no van a generar problemas en otras clases que hagan uso de esta?

- Localizar puntos de variación o inestabilidad (métodos para los que se prevea que sus clases de parámetros de entrada o su clase devuelta puedan cambiar en un futuro), crear una interfaz que esas clases pasadas como parámetro o devueltas como resultado implementen, y sustituir las clases concretas por las interfaces en la declaración de esos métodos correspondientes

```
public class ClaseA1 {…}

public class Operaciones {
   public void Operacion1(ClaseA1 entrada) {…}
}
```



```
public interface InterfaceA {…}
public class ClaseA1 {…}
public class ClaseA2 {…}

public class Operaciones {
   public void Operacion1(InterfaceA entrada) {…}
}
```

Principios GRASP: Pure fabrication

Pregunta: ¿Qué clase debe encargarse de hacer una operación cuando no se quieren incumplir los principios High Cohesion ni Low Coupling, pero ninguno de los demás principios nos da una respuesta adecuada?

- En estos casos, hay que crear una nueva clase que no represente ningún objeto del dominio del problema, que simplemente exista para agrupar una serie de métodos relacionados entre sí que son necesarios al margen de los que sí tienen que ver con el dominio del problema.

- En DDD (Domain-Driven Design), se definen los Servicios de Dominio (Domain Services), que no contienen código relacionado directamente con el problema, pero se necesitan en algún punto para que el programa funcione: configuración, conexión con BD, etc…

04

Patrones de diseño

Patrones de diseño

En este caso, no hay un único autor, sino cuatro: the Gang of Four (GoF)

- Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides

Design Patterns: Elements of Reusable Object-Oriented Software (1994)

Patrones de diseño

Se describen 23 patrones de diseño, agrupados según sirvan para

- crear nuevos objetos (Creational patterns)
- definir la estructura de un objeto o la comunicación entre objetos (Structural patterns)
- Ejecutar código o seleccionar qué código ejecutar (Behavioural patterns)

Recurso externo: canal de BettaTech en Youtube, lista de reproducción ‘Patrones de diseño’

Patrones de creación

- Factory method: lo veremos en detalle
- Abstract Factory: para abstraer distintos tipos de Factory de los del punto anterior
- Builder: para abstraer fases equivalentes de ejecución (mismos esqueletos)
- Prototype: para crear clones (copias por valor) de objetos (copia normal por referencia)
- Singleton: para crear una única instancia en un programa. Útil en programación paralela

Una idea aproximada: Simple factory

- La idea es crear una clase nueva con un método que se encargue de crear la clase deseada, en lugar de invocar directamente al constructor correspondiente

- Se aplica allí donde la creación de un objeto pueda ser compleja, o simplemente queramos ocultar al usuario la invocación al constructor correspondiente

- Hay varias formas de conseguirlo:
  - Pasar clase (o identificador único de la clase) por parámetro, y un switch-case para llamar a un constructor u otro
  - Crear un método distinto por clase a crear, y que invoquen al método correspondiente para crear la clase deseada (incluso pueden existir distintos métodos para crear objetos de una misma clase)

Una idea aproximada: Simple factory

- Pasar clase (o identificador único de la clase) por parámetro, y un switch-case para llamar a un constructor u otro:

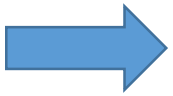
```
public class ClaseA {…}

public class ClaseA1: ClaseA {…}

public class ClaseA2: ClaseA {…}
```

```
public class CAFactory {
    public ClaseA nuevo(string version) {
        if (versión == “A1”) return new ClaseA1();
        else if (versión == “A2”) return new ClaseA2();
    }
}
```

```
public class Operaciones {
    public void Ejecutar() {
        CAFactory caf = new CAFactory();
        ClaseA nuevoClaseA = caf.nuevo(“A1”);
    }
}
```



```
public class Operaciones {
    public void Ejecutar() {
        CAFactory caf = new CAFactory();
        ClaseA nuevoClaseA = caf.nuevo(“A2”);
    }
}
```

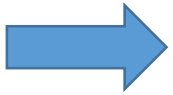
Patrones de creación: Factory

- ¡¡¡AQUÍ ME HE QUEDADO!!! Factory / Factory method

- Si hoy tenemos una línea de código donde creamos una instancia de una determinada clase, y mañana necesitamos crear otra clase diferente, tenemos que cambiar esa línea. Pero eso no está bien según el principio Open/Closed de SOLID…

```
public class ClaseA1 {…}

public class Operaciones {
    public void Ejecutar()
    {
        ClaseA1 nuevoClaseA = new ClaseA1();
    }
}
```



```
public class ClaseA1 {…}
public class ClaseA2 {…}

public class Operaciones {
    public void Ejecutar()
    {
        ClaseA2 nuevoClaseA = new ClaseA2();
    }
}
```

Patrones de creación: Factory method

- La solución pasa por crear una nueva clase con un método que permita crear un nuevo objeto de ClaseA1 o de ClaseA2, según necesitemos. El patrón Factory method plantea una manera que usa herencia (o interfaz común) y polimorfismo:

```
public class ClaseA {…}
```

```
public class ClaseA1: ClaseA {…}
```

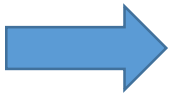
```
public class ClaseA2: ClaseA {…}
```

```
public class CA1Factory {
    public override ClaseA nuevo() { return new ClaseA1(); }
}
```

```
public class CA2Factory {
    public override ClaseA nuevo() { return new ClaseA2(); }
}
```

```
public abstract class CAAbstractFactory {
    public abstract ClaseA nuevo();
}
```

```
public class Operaciones {
    public void Ejecutar() {
        CA1Factory caf = new CA1Factory();
        ClaseA nuevoClaseA = caf.nuevo();
    }
}
```



```
public class Operaciones {
    public void Ejecutar() {
        CA2Factory caf = new CA2Factory();
        ClaseA nuevoClaseA = caf.nuevo();
    }
}
```

Patrones de estructura

- Adapter: lo veremos en detalle
- Bridge: interfaz para varias abstracciones, cada una con varias implementaciones
- Composite: interfaz común para elemento y conjunto de elementos
- Decorator: interfaz común para aplicar distintas operaciones a un mismo objeto
- Facade: interfaz para trabajar con distintos subsistemas intercambiables
- Flyweight: ahorra memoria al guardar atributos comunes en una clase aparte
- Proxy: una clase intermedia transparente que hace algo antes de dejar seguir

Patrones de estructura: Adapter

- Adapter

- Si tenemos una clase con un método al que queremos llamar, pero queremos hacerlo con una signatura diferente (diferente nombre o diferentes parámetros), tenemos que crear otra clase con un método con la signatura deseada que llame al método original

```
public class ClaseA {
    public void nombreOriginal() {...}
}
public class Operaciones {
    public void Ejecutar() {
        ClaseA nuevoClaseA = new ClaseA();
        nevoClaseA.nombreDeseado();
    }
}
```

Patrones de estructura: Adapter

- Adapter

- Si tenemos una clase con un método al que queremos llamar, pero queremos hacerlo con una signatura diferente (diferente nombre o diferentes parámetros), tenemos que crear otra clase con un método con la signatura deseada que llame al método original

```
public class ClaseA {
   public void nombreOriginal() {…}
}
```

```
public class Operaciones {
   public void Ejecutar() {
      ClaseA nuevoClaseA = new ClaseA();
      ClaseAAdapter ncaa = new ClaseAdapter();
      ncaa.nombreDeseado();
   }
}
```

```
public class ClaseAAdapter {
   public ClaseA ca { get; set; }
   public ClaseAAdapter(ClaseA caParam) {
      ca = caParam;
   }
   public void nombreDeseado() {
      ca.nombreOriginal();
   }
}
```

integra

Patrones de comportamiento

- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy: lo veremos con detalle
- Template method
- Visitor

Patrones de comportamiento: Strategy

- Strategy

- Si tenemos un método que incluye una parte que se puede querer implementar de varias formas distintas, y el resto del método es igual en todos los casos, se puede abstraer ese trozo de método mediante el patrón Strategy

```
public class Operaciones {
   public void Ejecutar() {
      …
   OrdenarListaPorNombre();
      …
   }
}
```

```
public class Operaciones {
   public void Ejecutar() {
      …
      OrdenarListaPorEdad();
      …
   }
}
```

Patrones de comportamiento: Strategy

- Strategy

- Si tenemos un método que incluye una parte que se puede querer implementar de varias formas distintas, y el resto del método es igual en todos los casos, se puede abstraer ese trozo de método mediante el patrón Strategy

```
public class Operaciones {
   public IOrdenacion io { get; set; }
   public Operaciones (IOrdenacion ioParam) {
      io = ioParam;
   }
   public void Ejecutar() {
      …
      io.OrdenarLista();
      …
   }
}
```

```
public class OrdenarPorNombre: IOrdenacion {
   public void OrdenarLista() {
      // código para ordenar por nombre
   }
}
```

```
public class OrdenarPorEdad: IOrdenacion {
   public void OrdenarLista() {
      // código para ordenar por edad
   }
}
```

```
public interface IOrdenacion {
   void OrdenarLista();
}
```

Ejemplo de reusabilidad

Enunciado inicial: Mostrar por pantalla todos los elementos de una lista de dos personas: trabajador y desempleado

…

Pasan 6 meses, y el enunciado para esa misma funcionalidad cambia: ahora el cliente quiere guardar en fichero todos los elementos de un array de dos personas: trabajador y jubilado

¿Qué código sería adecuado para poder abstraer el tipo de persona, el tipo de almacenamiento y el tipo de salida?

MUCHAS GRACIAS

Efor internet y tecnología

integra

Pasión por la innovación

www.integratecnologia.es

Tareas pendientes

- Añadir un ejemplo por cada patrón de diseño del GoF, aplicado sobre el ejercicio 17 GestionTrabajadores (si es factible)