PREGRADO

UNIDAD 1 | OVERVIEW

SOLID PRINCIPLES

SI720 | Diseño y Patrones de Software

Al finalizar la unidad de aprendizaje, el aplica los principios SOLID y buenas prácticas de arquitectura de información y datos para el diseño de aplicaciones, bajo el paradigma orientado a objetos.

AGENDA

INTRO

SOLID

Síntomas de Rotting Design

Rigidity

Fragility

Immobility

Viscosity

Rigidity (Rigidez)

Tendencia del software a tornar difícil la realización de cambios, incluso sencillos, generando una cascada de cambios en dependencias.

Fragility (Fragilidad)

Tendencia del software a quebrarse en muchos puntos cada vez que se realiza un cambio, a menudo en áreas sin relación conceptual con el área modificada.

Immobility (Inmobilidad)

Incapacidad de reutilizar software de otros proyectos o partes del mismo proyecto, dada la gran cantidad de dependencias que impiden aislar el módulo.

Viscocity (Viscosidad)

Puede existir en el diseño y en el entorno.

Puede haber más de una manera para afrontar un cambio, algunas preservan el diseño, otras no (hacks).

Cuando los caminos que preservan el diseño son más difíciles de seguir, se dice que el nivel de viscocity del diseño es alto.

Viscocity en el entorno se produce en entornos de desarrollo lentos e ineficientes, provocando el uso de caminos que eviten verificaciones, sin preocuparse si se preserva el diseño.

Degradación del diseño

Los requisitos cambian de forma no prevista por el diseño inicial.

Los ingenieros a cargo no están familiarizados con la filosofía del diseño original.

Los requisitos son cambiantes por naturaleza, hay que encontrar formas de que el diseño sea resistente y no se deteriore.

Degradación del diseño

¿Qué clase de cambios provocan deterioro en el diseño?

Aquellos que introducen dependencias nuevas y no planificadas.

¿Qué se requiere?

Dependency Management

Crear Firewalls para las dependencias (principios y patrones).

SOLID software design principles

	Principle	Description
S	Single responsibility	Una clase debería tener una y solo una razón para cambiar, lo cual quiere decir que debería tener solo una función.
O	Open/closed	Los objetos de software deberían ser abiertos a la extension pero cerrados a la modificación.
L	Listkov substitution	Los objetos del mismo tipo deberían poder reemplazarse por otros de la misma categoría sin alterar la función del programa.
I	Interface segregation	Ningún cliente debería ser forzado a depender de métodos que no usa. Las interfaces del programa siempre deberían mantenerse pequeñas y separadas entre sí.
D	Dependency inversion	Los módulos de alto nivel no deberían depender de los módulos de bajo nivel, sin embargo ambos deberían depender de abstracciones.

Single Responsibility Principle

El primer SOLID design principle, representado por la letra **S**.

Establece que, en una aplicación bien definida, cada clase debería tener sólo una única responsabilidad.

Responsabilidad se entiende como tener una sola razón para cambiar.

Cuando una clase tiene más de una responsabilidad, cualquier cambio en la funcionalidad podría afectar a otras.

Asegurarse que cada módulo encapsule solo una responsabilidad, ahorra testing time y crea una arquitectura que sea pueda mantener mejor.

Single Responsibility Principle

```
class Book {

   String title;
   String author;

   String getTitle() {
        return title;
   }
   void setTitle(String title) {
        this.title = title;
   }
   String getAuthor() {
        return author;
   }
   void setAuthor(String author) {
        this.author = author;
   }
   void searchBook() {...}

}
```

Clase **Book**

Responsabilidades

Establecer data de libro

Buscar libro en el inventario

Problema:

Dos responsabilidades

Single Responsibility Principle

```
class Book {

    String title;
    String author;

    String getTitle() {
        return title;
    }
    void setTitle(String title) {
        this.title = title;
    }
    String getAuthor() {
        return author;
    }
    void setAuthor(String author) {
        this.author = author;
    }

}
```

```
class InventoryView {

    Book book;

    InventoryView(Book book) {
        this.book = book;
    }

    void searchBook() {...}

}
```

Solución:

Desacoplar responsabilidades

Open/Closed Principle

Las clases, módulos, microservicios u otra unidad de código debería estar abierta para extensión, pero cerrara para las modificaciones.

Extensión vía subclases e interfaces.

No se debería modificar clases, posibilidad de resultados inesperados.

Open/Closed Principle

Soporte de descuento en Cookbook

```
class CookbookDiscount {

   String getCookbookDiscount() {

      String discount = "30% between Dec 1 and 24.";

      return discount;
   }

}

class DiscountManager {

   void processCookbookDiscount(CookbookDiscount discount)     {...}

}
```

Open/Closed Principle

```
class BiographyDiscount {

    String getBiographyDiscount() {

        String discount = "50% on the subject's birthday.";

        return discount;

    }
}

class DiscountManager {

    void processCookbookDiscount(CookbookDiscount discount) {...}

    void processBiographyDiscount(BiographyDiscount discount) {...}

}
```

Luego,

Descuento en Biography

Es necesario modificar

DiscountManager

La modificación

Requiere testing y deployment

Open/Closed Principle

```
public interface BookDiscount {

String getBookDiscount();

}


class CookbookDiscount
        implements BookDiscount {

   @Override
   public String getBookDiscount() {
   String discount =
        "30% between Dec 1 and 24.";

   return discount;
   }

}
```

```
class BiographyDiscount
        implements BookDiscount {

   @Override
   public String getBookDiscount() {
      String discount =
       "50% on the subject's birthday.";

      return discount;
   }

}

class DiscountManager {

   void processBookDiscount(
        BookDiscount discount) {...}
}
```

Solución:

Refactoring, flexibilidad en Discount

Liskov Substitution Principle

Un objeto de una superclase debería ser reemplazable por objetos de sus subclases sin provocar issues en la aplicación.

Un hijo no debería modificar las características de su clase padre.

Liskov Substitution Principle

```
class BookDelivery {

   String titles;
   int userID;

   void getDeliveryLocations() {...}
}

class HardcoverDelivery extends BookDelivery {

   @Override
   void getDeliveryLocations() {...}

}

class AudiobookDelivery extends BookDelivery {

   @Override
   void getDeliveryLocations() {
       /* can't be implemented */}
}
```

BookDelivery, informa a clientes sobre ubicaciones para recoger órdenes

HardcoverDelivery necesita personalizar método.

Audiobook, getDeliveryLocations, what?

Liskov Substitution Principle

```
class BookDelivery {

   String title;
   int userID;

}

class OfflineDelivery
        extends BookDelivery {

   void getDeliveryLocations() {...}

}

class OnlineDelivery
        extends BookDelivery {

   void getSoftwareOptions() {...}

}
```

```
class HardcoverDelivery
        extends OfflineDelivery {

   @Override
   void getDeliveryLocations() {...}

}

class AudiobookDelivery
        extends OnlineDelivery {

   @Override
   void getSoftwareOptions() {...}

}
```

Se agrega una capa a la jerarquía,

diseño flexible

Interface Segregation Principle

Los clientes no deberían estar obligados a depender de métodos que no utilizan, es decir, las interfaces no deberían incluir demasiadas funcionalidades (*fat interfaces*).

Interface Segregation Principle

```
public interface BookAction {

    void seeReviews();
    void searchSecondhand();
    void listenSample();

}
```

Book soporta interacciones.

Problema:

HardcoverUI y

AudiobookUI dependen de

métodos que no usan

```
class HardcoverUI implements BookAction {

    @Override
    public void seeReviews() {...}

    @Override
    public void searchSecondhand() {...}

    @Override
    public void listenSample() {...}

}

class AudiobookUI implements BookAction {

    @Override
    public void seeReviews() {...}

    @Override
    public void searchSecondhand() {...}

    @Override
    public void listenSample() {...}

}
```

Interface Segregation Principle

```
public interface BookAction {

   void seeReviews();

}

public interface HardcoverAction
        extends BookAction {

   void searchSecondhand();

}

public interface AudioAction
        extends BookAction {

   void listenSample();

}
```

```
class HardcoverUI
        implements HardcoverAction {

   @Override
   public void seeReviews() {...}

   @Override
   public void searchSecondhand() {...}

}

class AudiobookUI
        implements AudioAction {

   @Override
   public void seeReviews() {...}

   @Override
   public void listenSample() {...}

}
```

Solución:

Segregar Action

Dependency Inversion Principle

Evitar código con alto acoplamiento, lo cual debilita la aplicación.

Desacoplar high-level & low-level clases.

High-level classes dependientes de abstracciones en vez de implementaciones concretas.

Dependency Inversion Principle

Poner Book favorito en Shelf

```
class Book {

   void seeReviews() {...}
   void readSample() {...}

}

class Shelf {

   Book book;

   void addBook(Book book) {...}
   void customizeShelf() {...}

}
```

Dependency Inversion Principle

```
class DVD {

   void seeReviews() {...}
   void watchSample() {...}

}
```

Luego se requiere permitir

agregar DVDs

Problema:

Modificar Shelf

Dependency Inversion Principle

```
class Shelf {

    Product product;

    void addProduct(Product product) {...}

    void customizeShelf() {...}

}
```

Solución:

Adicionar Product, capa de abstracción.

```
public interface Product {

    void seeReviews();
    void getSample();

}

class Book implements Product {

    @Override
    public void seeReviews() {...}

    @Override
    public void getSample() {...}

}

class DVD implements Product {

    @Override
    public void seeReviews() {...}

    @Override
    public void getSample() {...}

}
```

RESUMEN

Recordemos

Rotting Design

SOLID

REFERENCIAS

Para profundizar

https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf
https://raygun.com/blog/solid-design-principles/

PREGRADO

Ingeniería de Software

Escuela de Ingeniería de Sistemas y Computación | Facultad de Ingeniería

UPC
Universidad Peruana
de Ciencias Aplicadas

Prolongación Primavera 2390,
Monterrico, Santiago de Surco
Lima 33 - Perú
T 511 313 3333
https://www.upc.edu.pe

exígete, innova

LAUREATE
INTERNATIONAL
UNIVERSITIES