

Análisis y diseño de aplicaciones I



UT5 – Patrones de diseño

1

Agenda



- Revisión SOLID
 - Single Responsibility Principle
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle
- Anti-Patrones
 - The Blob
 - Lava Flow
 - Golden Hammer
 - Spaghetti Code
 - Cut-and-Paste Programming
 - Monster commit
 - Tester driven development

2

1

Patrón



- Es un concepto en **ingeniería de software** que se refiere a una solución **probada y reutilizable** a un problema común dentro de un **contexto** específico en el diseño de software. Los patrones de software **NO** son fragmentos de código que pueden ser copiados y pegados en un programa, sino más bien guías generales que describen cómo abordar ciertos problemas y situaciones.
- Los patrones de software pueden ayudar a:
 1. **Resolver problemas comunes:** Al utilizar soluciones que han sido probadas en múltiples proyectos, los desarrolladores pueden evitar **reinventar la rueda**.
 2. **Mejorar la comunicación** entre desarrolladores: Los patrones de software tienen **nombres estándar**. Cuando los desarrolladores usan estos nombres en sus discusiones, otros desarrolladores familiarizados con esos patrones entienden rápidamente la solución propuesta.
 3. **Hacer que el código sea más mantenible y flexible:** Los patrones a menudo enfatizan la creación de código que es fácil de modificar y extender, lo que es beneficioso a largo plazo.

3

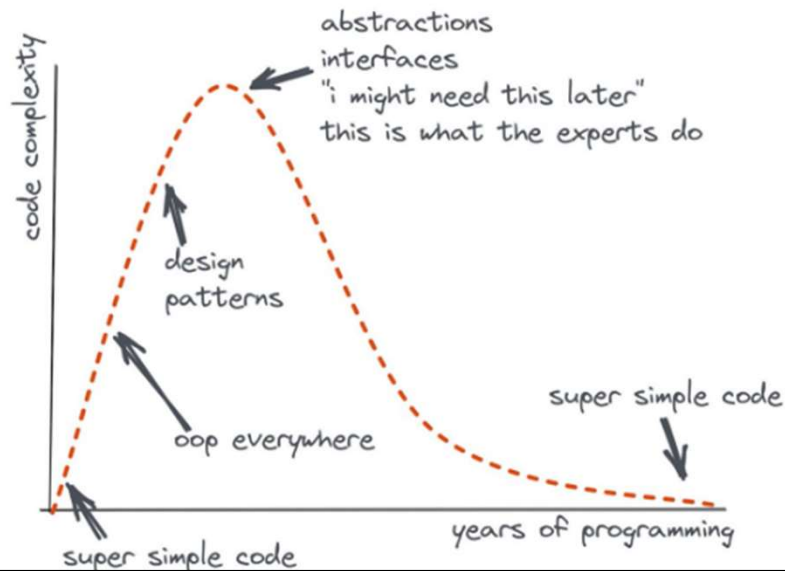
Patrón



1. **Patrones de Creación:** Se centran en la creación de objetos o clases.
2. **Patrones Estructurales:** Tratan de cómo se componen los objetos para formar estructuras más grandes.
3. **Patrones de Comportamiento:** Se centran en la interacción y responsabilidades entre objetos y clases.
4. **Patrones de Arquitectura:** Estos patrones están a un nivel más alto que los patrones de diseño y tratan con la arquitectura global de una aplicación. Un ejemplo común es el patrón MVC (Modelo-Vista-Controlador), que separa la lógica de negocio, la interfaz de usuario y la entrada del usuario en componentes independientes.
5. **Patrones de Managment:** Prácticas, estrategias y técnicas que ayudan a gestionar de manera efectiva un equipo, un proyecto o una organización. Estos patrones se pueden aplicar en diversos ámbitos, incluido el desarrollo de software.
 - Es importante notar que aunque los patrones de software ofrecen muchas ventajas, **no siempre son la solución adecuada para cada problema**, y su uso inadecuado puede resultar en **complicaciones innecesarias**. Por lo tanto, es esencial entender bien los patrones y **aplicarlos de manera juiciosa**.

4

2



5

Agenda

- Revisión SOLID
 - Single Responsibility Principle
 - Open-Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle
- Anti-Patrones
 - The Blob
 - Lava Flow
 - Golden Hammer
 - Spaghetti Code
 - Cut-and-Paste Programming
 - Monster commit
 - Tester driven development

6

SOLID



SOLID es un **conjunto de principios fundamentales** en ingeniería de software. El acrónimo representa los cinco principios básicos de la programación orientada a objetos y diseño, que incluyen:

- Single Responsibility (Responsabilidad Única),
- Open-Closed (Abierto-Cerrado),
- Liskov Substitution (Sustitución de Liskov),
- Interface Segregation (Segregación de Interfaces) y
- Dependency Inversion (Inversión de Dependencias).

Cuando se aplican en conjunto, estos principios ayudan a los desarrolladores a **crear sistemas que sean más fáciles de mantener y ampliar con el tiempo**. SOLID proporciona directrices para **evitar malos diseños** en el desarrollo de software, permitiendo que el código fuente sea refactorizado hasta que sea legible y extensible.

7

Single Responsibility Principle (SRP) UCU

- Establece que una clase debe **tener solo una razón para** cambiar, ya que cada responsabilidad es un posible cambio.
- Cuando **una clase tiene más de una responsabilidad**, las responsabilidades se **acoplan**, lo que puede conducir a diseños frágiles que pueden romperse de formas inesperadas cuando cambian. Por ejemplo, una clase Rectángulo que maneja la representación gráfica de un rectángulo y el cálculo de su área, sería utilizada de manera diferente en una aplicación gráfica y una de geometría computacional.
- La solución es dividir estas responsabilidades en dos clases distintas para evitar problemas.
- En el contexto del SRP, una **responsabilidad** se define como "**una razón para cambiar**".
 - Si puedes pensar en más de un motivo para cambiar una clase, entonces esa clase tiene más de una responsabilidad.
- En conclusión, el SRP es uno de los principios más sencillos y difíciles de aplicar correctamente. La esencia del diseño de software se basa en identificar y separar las responsabilidades.

8

Single Responsibility Principle (SRP) UCU

```
public class Rectangulo
{
    public double Largo { get; set; }
    public double Ancho { get; set; }

    public double Area()
    {
        return Largo * Ancho;
    }

    public void Dibujar()
    {
        // Logica para dibujar el rectángulo
    }
}
```

9

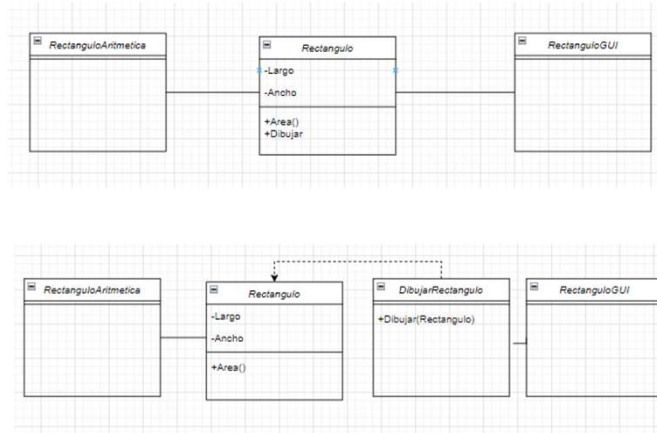
Single Responsibility Principle (SRP) UCU

```
public class Rectangulo
{
    public double Largo { get; set; }
    public double Ancho { get; set; }

    public double Area()
    {
        return Largo * Ancho;
    }
}

public class DibujarRectangulo
{
    public void Dibujar(Rectangulo rectangulo)
    {
        // Logica para dibujar el rectángulo
    }
}
```

10



11

The Open-Closed Principle

- Establece que las entidades de software (clases, módulos, funciones, etc.) deben estar **abiertas para su extensión, pero cerradas para su modificación.**
- OCP puede abordar problemas comunes asociados con los "malos" diseños de software,
 - fragilidad
 - rigidez
 - imprevisibilidad
 - falta de reutilización.
- En otras palabras, un diseño que sigue el OCP puede **soportar cambios en los requisitos** a través de la extensión de módulos (agregando nuevo código), en lugar de cambiar el código existente que ya funciona.
- Para que un módulo cumpla con el OCP, debe tener dos atributos principales:
 1. **"Abierto para extensión"**: esto significa que el comportamiento del módulo puede extenderse para adaptarse a los cambios en los requisitos de la aplicación o para satisfacer las necesidades de nuevas aplicaciones.
 2. **"Cerrado para modificación"**: esto significa que el código fuente de un módulo es inviolable y nadie puede hacer cambios en él. *Stalin*

12

The Open-Closed Principle



- Las variables globales/públicas, violan el principio de abierto-cerrado. Ningún módulo que dependa de una variable global puede estar cerrado contra cualquier otro módulo que pueda escribir en esa variable.
 - Cualquier módulo que utilice la variable de una manera que los otros módulos no esperen, romperá esos otros módulos. Por lo tanto, es demasiado arriesgado tener muchos módulos sujetos al capricho de un módulo mal comportado.

13

The Open-Closed Principle



- La identificación de tipos en tiempo de ejecución (RTTI) es peligrosa, ya que a menudo puede violar el principio de abierto-cerrado.
- La diferencia entre un mal y un buen uso de RTTI radica en si el código necesita ser cambiado cada vez que se deriva un nuevo tipo de objeto.

14

The Open-Closed Principle



```
class Shape {};  
class Square: public Shape {  
    private: Point itsTopLeft;  
    double itsSide;  
    friend DrawSquare(Square * );  
};  
class Circle: public Shape {  
    private: Point itsCenter;  
    double itsRadius;  
    friend DrawCircle(Circle * );  
};  
  
void DrawAllShapes(Set < Shape * > & ss) {  
    for (Iterator < Shape * > i(ss); i; i++) {  
        Circle * c = dynamic_cast < Circle * > ( * i);  
        Square * s = dynamic_cast < Square * > ( * i);  
        if (c)  
            DrawCircle(c);  
        else if (s)  
            DrawSquare(s);  
    }  
}
```

15

The Open-Closed Principle



```
class Shape {  
    public: virtual void Draw() const = 0;  
};  
class Square: public Shape {  
    // as expected.  
};  
void DrawSquaresOnly(Set < Shape * > & ss) {  
    for (Iterator < Shape * > i(ss); i; i++) {  
        Square * s = dynamic_cast < Square * > ( * i);  
        if (s)  
            s->Draw();  
    }  
}
```

16

Liskov



- *“Subclasses should be substitutable for their base classes.”*
- Los principales mecanismos detrás del principio de abierto-cerrado son la abstracción y el polimorfismo. En lenguajes como C# uno de los mecanismos clave que soporta la abstracción y el polimorfismo es la herencia.
- Una clase B es una subclase de la clase A, entonces debe ser posible usar B donde sea que A se espere sin cambiar el comportamiento del programa.
- Un ejemplo clásico de violación de LSP en C# sería el siguiente:

17

Liskov



```
public class Rectángulo {  
    public virtual int Ancho {  
        get;  
        set;  
    }  
    public virtual int Alto {  
        get;  
        set;  
    }  
  
    public int Área() {  
        return Ancho * Alto;  
    }  
}
```

```
public class Cuadrado: Rectángulo {  
    public override int Ancho {  
        get {  
            return base.Ancho;  
        }  
        set {  
            base.Ancho = base.Alto = value;  
        }  
    }  
  
    public override int Alto {  
        get {  
            return base.Alto;  
        }  
        set {  
            base.Alto = base.Ancho = value;  
        }  
    }  
}
```

18

Liskov



```
public void SetDimensiones(Rectángulo rectángulo)
{
    rectángulo.Ancho = 5;
    rectángulo.Alto = 4;
}
```

Si pasas un Cuadrado a este método, al final tendrás un cuadrado con lado de longitud 4, en lugar de un rectángulo con ancho 5 y alto 4, lo cual viola el Principio de Sustitución de Liskov.

19

Interface Segregation Principle

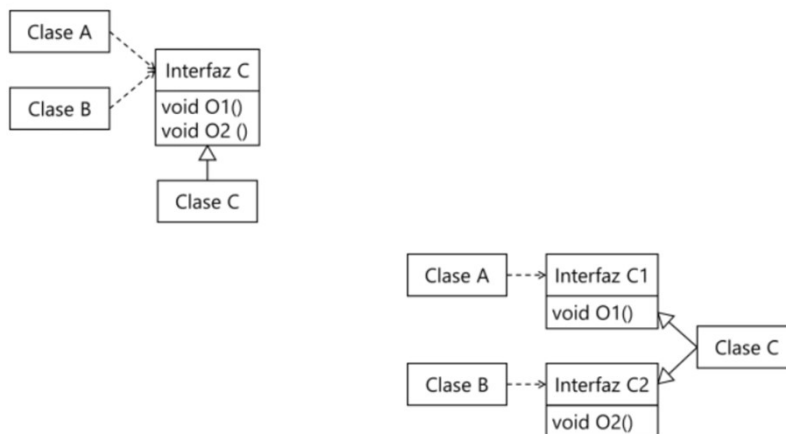
- Los clientes no **deben ser forzados a depender de tipos que no usan**. En otras palabras, es mejor tener muchas interfaces específicas que una sola interface general.
- Este principio tiene como objetivo reducir los problemas que pueden surgir debido a los cambios en las clases que dependen de las interfaces de gran tamaño. Los clientes de una interface que contiene métodos que no necesitan aún se ven afectados cuando la interface cambia.

20

Interface Segregation Principle

- Supongamos que tienes una interface IAve que tiene métodos para volar, nadar y comer. Ahora bien, no todas las aves pueden volar o nadar, por lo que si tuvieras una clase Pinguino que implementa la interface IAve, tendrías que implementar un método de volar que realmente no tiene sentido para un pingüino.
- Según el ISP, sería mejor tener tres interfaces separadas: IAveVoladora, IAveNadadora e IAveComedora. De esta forma, Pinguino podría implementar solo IAveNadadora e IAveComedora, y no estaría forzado a implementar IAveVoladora. Esto hace que el diseño sea más flexible y coherente.

21



22

Dependency Inversion Principle

- Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
- Las abstracciones no deben depender de los detalles → Los detalles deben depender de las abstracciones.
- El propósito de este principio es ayudar a desacoplar el software. La idea es que cuando los módulos de alto nivel están fuertemente acoplados a los módulos de bajo nivel, se dificulta la reutilización y las modificaciones en los módulos de bajo nivel.

23

Dependency Inversion Principle

- Supón que tienes una clase Aplicación (módulo de alto nivel) que depende de una clase BaseDeDatos (módulo de bajo nivel).
- En lugar de que Aplicacion dependa directamente de BaseDeDatos, puedes introducir una interfaz abstracta IBaseDeDatos que separe los dos módulos. Aplicacion debería depender de IBaseDeDatos, y BaseDeDatos debería implementar IBaseDeDatos.
- Así, si decides cambiar la implementación de la base de datos en el futuro (por ejemplo, pasar de MySQL a PostgreSQL), solo necesitarás crear una nueva clase que implemente IBaseDeDatos. No tendrás que hacer ningún cambio en la clase Aplicacion, ya que su dependencia es con la abstracción IBaseDeDatos, no con una implementación específica.

24

```
public class CorreoElectronico {  
    public void EnviarCorreo(string  
    contenido) {  
        // Envía un correo electrónico  
    }  
}
```

```
public class Notificaciones  
{  
    private CorreoElectronico  
    _correoElectronico = new  
    CorreoElectronico();  
  
    public void  
    EnviarNotificacion(string  
    mensaje)  
    {  
  
        _correoElectronico.EnviarCorreo(  
        mensaje);  
    }  
}
```

25

```
public interface IMensaje  
{  
    void Enviar(string mensaje);  
}  
  
public class CorreoElectronico : IMensaje  
{  
    public void Enviar(string mensaje)  
    {  
        // Envía un correo electrónico  
    }  
}  
  
public class MensajeTexto : IMensaje  
{  
    public void Enviar(string mensaje)  
    {  
        // Envía un mensaje de texto  
    }  
}
```

```
public class Notificaciones  
{  
    private IMensaje _mensaje;  
  
    public Notificaciones(IMensaje  
    mensaje)  
    {  
        _mensaje = mensaje;  
    }  
  
    public void  
    EnviarNotificacion(string mensaje)  
    {  
        _mensaje.Enviar(mensaje);  
    }  
}
```

26

Tarea de Aplicación 1

SOLID



27

Agenda



- Anti-Patrones
 - The Blob
 - Lava Flow
 - Golden Hammer
 - Spaghetti Code
 - Cut-and-Paste Programming
 - Tester driven development

28

Antipatrón



- Es un término utilizado en ingeniería de software que hace referencia a una **solución comúnmente utilizada** para resolver un problema en el desarrollo de software, pero que **produce resultados contraproducentes o negativos**. Los antipatrones son a menudo el resultado de *la experiencia limitada, la falta de conocimiento de mejores prácticas, plazos apresurados o soluciones a corto plazo sin considerar las consecuencias a largo plazo*.
- Es importante destacar que lo que hace que una práctica sea un antipatrón no es simplemente que sea una mala solución, sino que es una solución que parece ser beneficiosa en la superficie o a corto plazo, pero que conduce a **problemas en el futuro**. Son errores que se cometen comúnmente en la industria.
- Características:
 1. **Ineficiencia:** Pueden hacer que el software sea más lento o consuma más recursos de los necesarios.
 2. **Dificultad de mantenimiento:** Hacen que el código sea más difícil de entender, modificar o extender.
 3. **Introducción de defectos:** Pueden introducir errores o hacer que sea más probable que ocurran problemas.
 4. **Complejidad innecesaria:** Añaden complejidad al diseño o al código sin un beneficio claro.

29

The Blob

- Definición
 - Una sola clase se encarga de la mayoría de las responsabilidades mientras que otras clases son simples contenedores de datos.
 - Es un diseño similar a la programación procedural en lugar de ser verdaderamente orientado a objetos.
- Cómo Reconocer The Blob
 - Una clase con un número extremadamente alto de atributos y métodos.
 - Falta de cohesión en los atributos y operaciones de la clase.
 - Las demás clases sirven principalmente para almacenar datos y tienen poca o ninguna lógica.



30

The Blob



- Consecuencias de Usar The Blob
 - Dificulta la mantenibilidad y comprensión del código.
 - Reduce la reusabilidad de las clases.
 - Puede afectar el rendimiento debido a la carga excesiva en una sola clase.
- Cómo Evitar o Resolver The Blob
 - Refactorizar: Dividir la clase gigante en clases más pequeñas y especializadas.
 - Redistribuir responsabilidades entre clases basándose en principios de diseño orientado a objetos, como el Principio de Responsabilidad Única.
 - Asegurar que cada clase tenga un propósito claro y bien definido.

31

Lava flow

- Definición: Refiere a piezas de código que quedan en el sistema, aunque ya no son útiles, y se vuelven difíciles de eliminar.
- Características:
 - Código que se mantiene a pesar de que ya no es necesario
 - Código que se ha vuelto obsoleto debido a cambios en el diseño o requisitos
 - Código que se evita modificar por miedo a romper funcionalidades existentes
- Cómo se forma:
 - Cambios rápidos en los requerimientos del proyecto
 - Falta de entendimiento o documentación del código existente
 - Temor a modificar código antiguo por posibles consecuencias en el funcionamiento del sistema



32

Lava flow



- Consecuencias:
 - Aumento de la complejidad del código
 - Reducción de la mantenibilidad
 - Inflación del tamaño del código, haciendo que el software sea más difícil de entender y modificar
- Cómo prevenir y solucionar Lava Flow
 - Documentar bien el código y los cambios realizados
 - Implementar revisiones de código periódicas
 - Refactorizar el código obsoleto
 - Eliminar código innecesario después de pruebas rigurosas y revisiones de código

33

Golden Hammer

- Definición: cuando un equipo de desarrollo o individuo depende demasiado de una herramienta o tecnología familiar, a expensas de posiblemente mejores alternativas.
- Características:
 - Sobreutilización de una herramienta o tecnología en particular.
 - Reluctancia para considerar o adoptar alternativas.
 - Creencia de que la herramienta familiar puede resolver todos los problemas.
- Cómo se forma:
 - Falta de conocimiento de otras herramientas y tecnologías.
 - Comodidad y familiaridad con una herramienta específica.
 - Resistencia al cambio o a aprender nuevas habilidades.



34

Golden Hammer



- Consecuencias:
 - Soluciones ineficientes o subóptimas.
 - Mayor costo y tiempo de desarrollo.
 - Limitación en la innovación y adaptabilidad del proyecto.
- Cómo prevenir y solucionar Golden Hammer
 - Educar al equipo sobre diferentes herramientas y tecnologías.
 - Realizar evaluaciones objetivas antes de seleccionar herramientas o tecnologías.
 - Fomentar un entorno abierto a la experimentación y aprendizaje continuo.

35

Spaghetti Code



- Definición: el código fuente de un programa tiene una estructura compleja y enredada, parecida a un plato de espaguetis, lo que lo hace difícil de leer, mantener y depurar.
- Características:
 - Falta de estructura y organización.
 - Uso excesivo de saltos incondicionales (GOTOs).
 - Difícil de seguir el flujo de control.
 - Ausencia de modularidad y encapsulación.
- Cómo se forma:
 - Desarrollo apresurado sin una arquitectura adecuada.
 - Falta de conocimiento de buenas prácticas de programación.
 - Modificaciones y parches sucesivos sin reconsiderar la estructura general.

Spaghetti Code



- Consecuencias:
 - Dificultad en mantenimiento y depuración.
 - Incremento en el riesgo de introducir errores.
 - Mayor costo y tiempo de desarrollo.
- Recomendaciones para evitar el Spaghetti Code:
 - Planificar y diseñar la arquitectura antes de escribir el código.
 - Seguir buenas prácticas de programación.
 - Realizar revisiones de código.
- Recomendaciones para refactorizar el Spaghetti Code:
 - Dividir el código en funciones y clases más pequeñas y coherentes.
 - Reemplazar saltos incondicionales con estructuras de control más legibles.
 - Mejorar la documentación y comentarios

37

Cut-and-Paste P

- Definición: se refiere a la práctica de copiar y pegar bloques de código dentro de una aplicación, en lugar de crear funciones o módulos reutilizables.
- Características:
 - Duplicación de código.
 - Falta de modularidad y abstracción.
 - Dificultad en la implementación de cambios en lógicas duplicadas.
- Cómo se forma:
 - Desarrollo apresurado o bajo presión de tiempo.
 - Falta de conocimiento de buenas prácticas de programación.
 - Tratar de evitar "reinventar la rueda" sin tener una adecuada reutilización de código.



38

Cut-and-Paste P

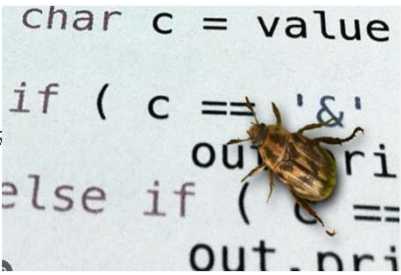


- Consecuencias
 - Dificultad en el mantenimiento y la corrección de errores.
 - Incremento en el riesgo de inconsistencias y errores lógicos.
 - Código menos legible y comprensible.
- Recomendaciones para evitar Cut-and-Paste Programming:
 - Pensar en modularidad y reutilización de código desde el inicio.
 - Crear funciones y clases para lógicas comunes.
- Recomendaciones para solucionar Cut-and-Paste Programming:
 - Identificar y eliminar la duplicación..
 - Refactor

39

Tester Driven Development



- O "Bug Driven Development", se refiere a un antipatrón en el que las **pruebas de software o los informes de errores dirigen el desarrollo de software**, en lugar de las necesidades del usuario o los requerimientos de las funcionalidades. Este enfoque puede dar lugar a una baja calidad del código y retrasos en la entrega del software.
- 
- Esto puede suceder si:
 - Las pruebas comienzan demasiado pronto
 - Los requerimientos no están completos
 - Los testers o desarrolladores son inexpertos
 - La gestión del proyecto es deficiente.
 - En estos casos, **los testers pueden terminar dictando cómo debería ser el software, lo que puede desviarse de lo que los usuarios realmente necesitan o quieren.**

40

Tester Driven Development



Para evitar caer en este antipatrón, es importante asegurarse de que:

1. Las pruebas de software comienzan en el momento adecuado: no demasiado pronto que no hay suficiente para probar, y no demasiado tarde que los errores no son detectados hasta que es muy costoso arreglarlos.
2. Los requerimientos están completos y bien definidos: los desarrolladores y testers deben tener una clara comprensión de lo que se supone que debe hacer el software.
3. Tanto los testers como los desarrolladores están adecuadamente capacitados: deben entender no sólo cómo realizar sus tareas, sino también cómo encajan en el proceso general de desarrollo de software.
4. La gestión del proyecto es eficaz: esto incluye planificar adecuadamente, comunicarse de manera eficaz, y asegurarse de que todos los miembros del equipo entienden sus roles y responsabilidades.

Mantener un enfoque en el valor del usuario y los requerimientos de las funcionalidades. Las pruebas de software son una herramienta importante para asegurar la calidad del software, pero no deben ser la única fuerza impulsora detrás del desarrollo del software.

41

Tarea de Aplicación 2 ANTIPATRONES



42

Bibliografía



- [https://github.com/ucudal/PII Principios Patron es](https://github.com/ucudal/PII_Principios_Patron_es)
- <https://webasignatura.ucu.edu.uy/mod/folder/view.php?id=579025>
- <https://twitter.com/raupach/status/1463490452980649987>

43

“Si te cansas, aprende a descansar, no a renunciar”



44