Análisis y diseño de aplicaciones I

UT5 – Patrones de diseño

1

Agenda

- Revisión clase pasada
  - SOLID
  - Antipatrones
- Patrones creacionales (Gang of Four)
  - Singleton
  - Factory
  - Prototype
  - Builder

2

SOLID (Repaso)

UCU

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:

- **S**ingle Responsibility (Responsabilidad Única)
- **O**pen-Closed (Abierto-Cerrado)
- **L**iskov Substitution (Sustitución de Liskov)
- **I**nterface Segregation (Segregación de Interfaces)
- **D**ependency Inversion (Inversión de Dependencias).

3

Antipatterns (Repaso)

UCU

- Describe una solución común a un problema que genera consecuencias únicamente negativas.
- Son el resultado de:
  - Falta de conocimiento de un manager o desarrollador, que no tiene suficiente experiencia resolviendo problemas de cierto tipo
  - De aplicarse un patrón perfectamente, pero en el contexto equivocado
- Se documentan con cierto cinismo, lo cual los hace bastante graciosos y fáciles de recordar.

4

Antipatterns (Repaso)

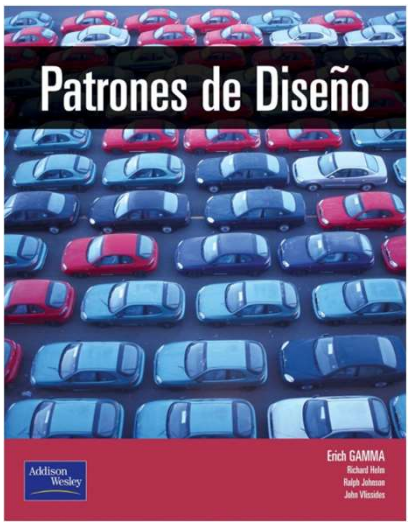
UCU

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

5

Design Patterns (Gang of Four)

UCU



Según *Design Patterns* existen 3 tipos de patrones:

- **Patrones Creacionales (Clase de hoy)**
- Patrones Estructurales
- Patrones de Comportamiento

6

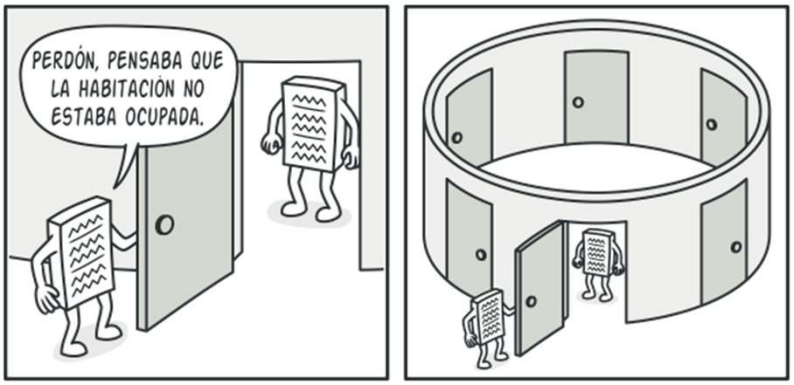
Design Patterns (Gang of Four)

- Son soluciones comunes a problemas relacionados con la creación de objetos en el diseño de software.
- Estos patrones se centran en proporcionar formas flexibles y reutilizables de crear y configurar objetos
- Los patrones creacionales abordan diferentes escenarios de creación de objetos, como garantizar una única instancia de una clase, crear objetos flexibles o construir objetos complejos paso a paso.

7

Singleton

Se utiliza para garantizar que una clase solo tenga una única instancia en todo el programa, proporcionando un punto de acceso global a esa instancia



8

Singleton

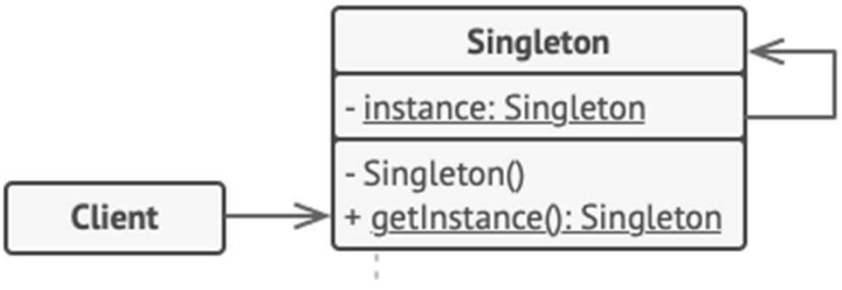
UCU

- Resuelve varios problemas:
  - Garantiza que una clase tenga una única instancia
  - Proporcionar un punto de acceso global a dicha instancia
  - El objeto Singleton solo se inicializa cuando se requiere por primera vez
- Contras:
  - Viola principio SRP: Genera instancias, pero también las limita
  - Puede enmascarar un mal diseño:
    - Los componentes del programa saben demasiado los unos sobre los otros.
    - Produce acoplamiento
    - Complica automatización de tests

9

Singleton (UML de Referencia)

UCU

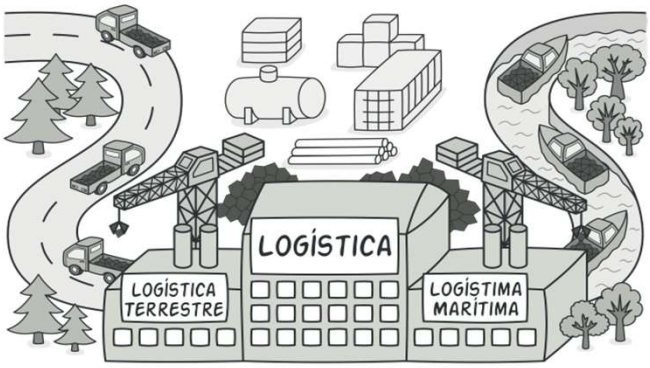


- Código de referencia: https://github.com/VanHakobyan/DesignPatterns/tree/master/Singleton

10

Factory Method

Se utiliza para encapsular la creación de objetos en una clase separada, permitiendo delegar la responsabilidad de la creación de objetos a la clase “Fábrica”



11

Factory Method

- Proporciona una interfaz para crear objetos en una superclase y permite a las subclases alterar el tipo de objetos que se desean crear.
- Separa el código de construcción de producto del código que hace uso del producto.
- Por ello, es más fácil extender el código de construcción de producto de forma independiente al resto del código

12

Factory Method (Usos)

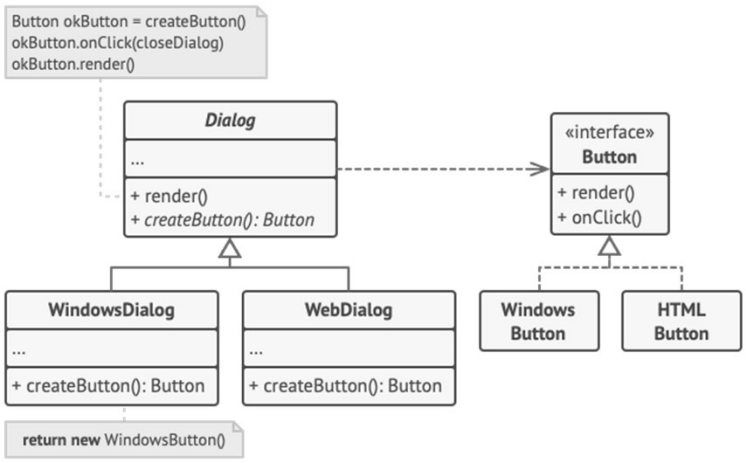
UCU

- Cuando se trabaja con jerarquías de clases y se necesita crear objetos polimórficos
- Cuando se busca desacoplar el código entre el cliente y las clases concretas
- Promueve la extensibilidad del código al permitir que nuevas subclases se agreguen fácilmente para crear nuevos tipos de objetos

13

Factory Method

UCU



14

Factory Method (Estructura)

- Todos los objetos a crear deben seguir la misma interfaz *"IProducto"*
- La clase *"Factory"* que define el comportamiento con un *"Factory Method"*
- A raíz de *"Factory"* van a existir subclases que van a heredar el comportamiento de clase padre
- Las subclases van a implementar el "Factory Method" con la lógica deseada, devolviendo un "IProducto"

15

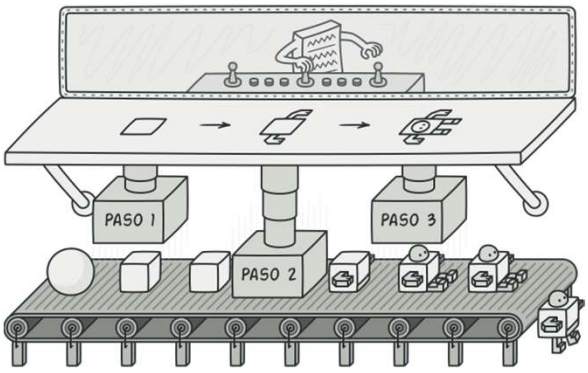
Factory Method

- Pros:
  - Evita un acoplamiento fuerte entre el creador y los objetos
  - Aplica OCP y SRP para su solución
- Contras:
  - Complejidad del código aumenta, el patrón exige la creación de clases nuevas.
- Código de referencia: https://github.com/VanHakobyan/DesignPatterns/tree/master/FactoryMethod

16

Builder

Permite construir objetos complejos paso a paso, permitiendo también producir distintos tipos y representaciones de un objeto empleando el mismo código de construcción.



17

Builder

- El Builder abstrae el proceso de construcción del objeto final de su representación interna.
- El Builder permite la creación de diferentes variaciones o configuraciones del objeto final
- Separa la lógica de construcción del objeto de la clase del objeto en sí.

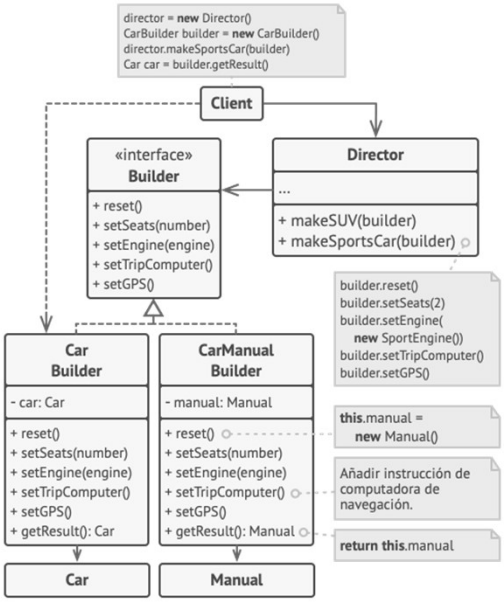
18

Builder (Usos)

- Es especialmente útil cuando existe un objeto **complejo** que requiere una inicialización laboriosa, paso a paso, de muchos campos y objetos anidados
- El patrón Builder facilita la modificación y ampliación del proceso de construcción
- Se suele puede aplicar cuando la construcción de varias representaciones de un producto requiera de pasos similares que sólo varían en los detalles.

19

Builder



20

Builder (Estructura)

- Debe existir una interfaz o clase abstracta *"Builder"* que defina el comportamiento con los métodos llamados "*steps*".
- Deben existir subclases de *"Builder"* que implementen su comportamiento.
- La clase *"Director"* define que subclases y pasos se ejecutan

21

Builder

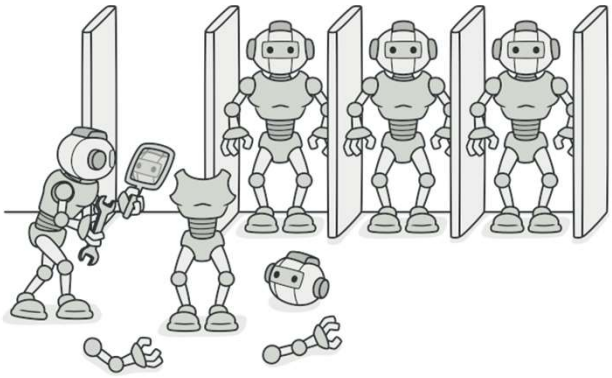
- Pros:
  - Construir objetos paso a paso, aplazar pasos de la construcción
  - Reutilizar el mismo código de construcción al construir varias representaciones de producto
  - Aplica SRP.
- Contras:
  - Complejidad del código aumenta, el patrón exige la creación de clases nuevas.
- Código de referencia: https://github.com/VanHakobyan/DesignPatterns/tree/master/Builder/Builder

22

Prototype

UCU

Se utiliza para crear nuevos objetos clonando un objeto existente, evitando la creación de objetos desde cero.



23

Prototype

UCU

- Permite crear nuevos objetos a través de la clonación de un objeto existente.
- Proporciona flexibilidad al permitir la creación de nuevos objetos con diferentes configuraciones o variantes sin tener que crear nuevas clases específicas para cada caso
- Permite modificar los atributos y comportamientos del objeto clonado sin afectar al objeto prototipo original.
- Algunos de los lenguajes lo implementan de forma nativa, por ejemplo: C# o C++

24

Prototype (Usos)

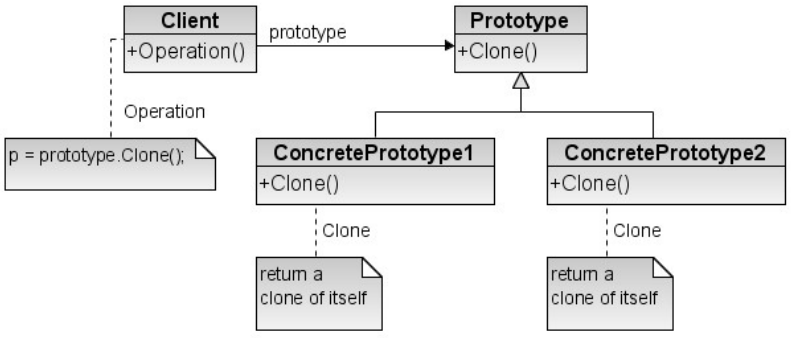
UCU

- Cuando se busca evitar dependencia de clases concretas de objetos que se necesiten copiar. (Por ejemplo, objetos pasados por códigos de terceras personas a través de una interfaz)
- Cuando se necesitan gestionar diferentes estados o versiones de un “mismo” objeto, puede ser una alternativa a la creación de subclases
- Si la creación de un objeto es costosa, Prototype puede mejorar la eficiencia. En lugar de repetir el proceso de creación completo cada vez, se puede clonar un objeto ya creado

25

Prototype

UCU



26

Prototype (Estructura)

UCU

- Debe existir una interfaz *"Prototype"* que declara los métodos de clonación, por ejemplo: *clonar()*
- Existe una clase que implementa *"Prototype"* que llamaremos *"ConcretePrototype"*
- La clase *"ConcretePrototype"* implementa el método *clonar()*
- Si el método clonar() copia las referencias de la clase *"ConcretePrototype",* el clonado se denomina cómo superficial
- De lo contrario, si copia los objetos de las referencias, estamos ante una clonación profunda

27

Prototype (Estructura)

UCU

- Pros:
  - Evitar inicialización repetida clonando prototipos prefabricados
  - Más fácil crear objetos complejos
  - Alternativa a la herencia al tratar con preajustes de configuración
- Contras:
  - Puede haber complicaciones con referencias circulares si emplea una clonación profunda

- Código de referencia: https://github.com/VanHakobyan/DesignPatterns/tree/master/Prototype

28

Caso de Estudio

" … En base a los patrones de diseño creacionales vistos, generar una funcionalidad con la capacidad de añadirle 2 filtros fotográficos vintage a una imagen, aplicando 1 a la vez y manteniendo un registro … "

29

Tarea de Aplicación 3

30

¿Preguntas?

UCU

31

Bibliografía

UCU

- https://medium.com/dise%C3%B1o-de-software/singleton-el-patr%C3%B3n-del-mal-f3fdab0e16a2
- https://refactoring.guru/es/design-patterns/creational-patterns
- https://refactoring.guru/es/design-patterns/builder
- https://refactoring.guru/es/design-patterns/singleton
- https://refactoring.guru/es/design-patterns/prototype
- https://refactoring.guru/es/design-patterns/abstract-factory
- https://github.com/VanHakobyan/DesignPatterns/tree/master
- Design Patterns (1994) Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside

32