UTN F r a
Avellaneda

Programación II

Clase 04

Programación Orientada a Objetos

Temario

UTNFra Avellaneda

Principios SOLID

- Origen
- Definiciones

Patrones de Diseño

- Usos
- Tipos
- Anti patrones

Patrones

- Diseño
- Sintaxis

01.

Principios SOLID

Principios SOLID

UTN F r a Avellaneda

Autores

- **Robert Martin (aka Uncle Bob)**
  https://es.wikipedia.org/wiki/Robert_C._Martin
- **Barbara Liskov**
  https://es.wikipedia.org/wiki/Barbara_Liskov
- **Bertrand Meyer**
  https://es.wikipedia.org/wiki/Bertrand_Meyer

Principios SOLID

Objetivo

Hacer que los diseños de software sean más legibles, flexibles y mantenibles en el tiempo.

Autores

Robert Martin (aka Uncle Bob) https://es.wikipedia.org/wiki/Robert_C._Martin

Barbara Liskov https://es.wikipedia.org/wiki/Barbara_Liskov

Bertrand Meyer https://es.wikipedia.org/wiki/Bertrand_Meyer

Problemas del Diseño

- **Rigidez**

  Un cambio en una parte de un programa rompe otra parte.

- **Fragilidad**

  Un error se presenta en partes no relacionadas.

- **Inmovibilidad**

  El código desarrollado no puede reutilizarse.

Principios SOLID

UTN F r a
Avellaneda

SOLID

- Single Responsibility
- Open-Closed
- Liskov Substitution
- Interface Segregation
- Dependency Invertion

Principios SOLID

Single Responsibility

Robert Martin

Una clase sólo debe tener una única razón para cambiar.

- **Responsabilidad**

  Una clase debe focalizarse en realizar una tarea que deberá estar bien encapsulada. Una tarea es una parte de una funcionalidad.

  Ante requerimientos nuevos es más fácil de identificar donde realizar las modificaciones.

Principios SOLID

Open-Closed

Bertrand Meyer

Abierto para extensiones, pero cerrado para modificaciones.

- **Open-Closed**

Agregar comportamiento a una clase (extender) sin la necesidad de modificarla.

Utilizar polimorfismo con herencia o, si se quiere evitar acoplamiento innecesario, interfaces.

Principios SOLID

Liskov Substitution

Barbara Liskov

Una clase debe poder ser reemplazada por una de sus subclases.

- **Restriciones en la herencia**

  Las precondiciones no pueden más fuertes en una subclase.

  Los tipos de retorno deber ser iguales.

  Los compiladores no lo detectan.

Principios SOLID

Interface Segregation

Robert Martin

Mantener interfaces lo más pequeñas posible.

- **Interfaces**

  Las interfaces son cerradas a los cambios.

  Separar comportamiento en múltiples interfaces favoreciendo la composición (y reusabilidad).

Principios SOLID

Dependency Inversion Robert Martin

El código debe depender de abstracción en lugar de otro código.

- **Flujo de control**

Las clases concretan sufren muchos cambios, mientras que intefaces y abstracciones, no.

Separar comportamiento en múltiples interfaces favoreciendo la composición (y reusabilidad).

02.

Patrones de Diseño

Patrones de Diseño

¿Qué son?

- Surgieron con la idea de realizar **desarrollo de software reusable**.
- Se los considera **buenas prácticas** (forma eficiente y escalable) para solucionar problemas conocidos y/o recurrentes.
- Se implementan en la etapa de diseño (antes del código).

Composición

- **Nombre**

  Abstracción.

- **Problema**

  Contexto en el que aplica el patrón.

- **Solución**

  Esquemática. Define elementos y sus relaciones y responsabilidades.

- **Consecuencias**

  popularidad, complejidad, extensibilidad

Patrones de Diseño

Tipos de Patrones

- **Creación**

  Administrar la creación de objetos.

- **Comportamiento**

  Interacción entre objetos.

- **Estructura**

  Componer objetos en estructuras más grandes y complejas si perder flexibilidad y reutilización de código.

Patrones de Diseño

Anti patrones

- **Reactivos**

  Soluciones que nacen cuando se manifiestan problemas particulares relacionados a un mal diseño.

- **No reutilizables**

  Al satisfacer un problema particular no es aplicable en otro lugar.

- **Mala práctica**

  Alimentan un crecimiento de código ineficiente.

02.A

Factory Method

Factory Method

Fábrica de Objetos

- **Objetivo**

  Se delega el proceso de creación de un objeto a otra clase u objeto (**fábrica**).

  Los objetos que retorna son del **mismo tipo**.

- **Uso**

  Se requiere crear un objeto que realice una acción.

  El cliente sólo tiene una especificación y no sabe (ni le importa) cómo se instancia ese objeto.

Factory Method

Fábrica de Objetos

- **Requisitos**

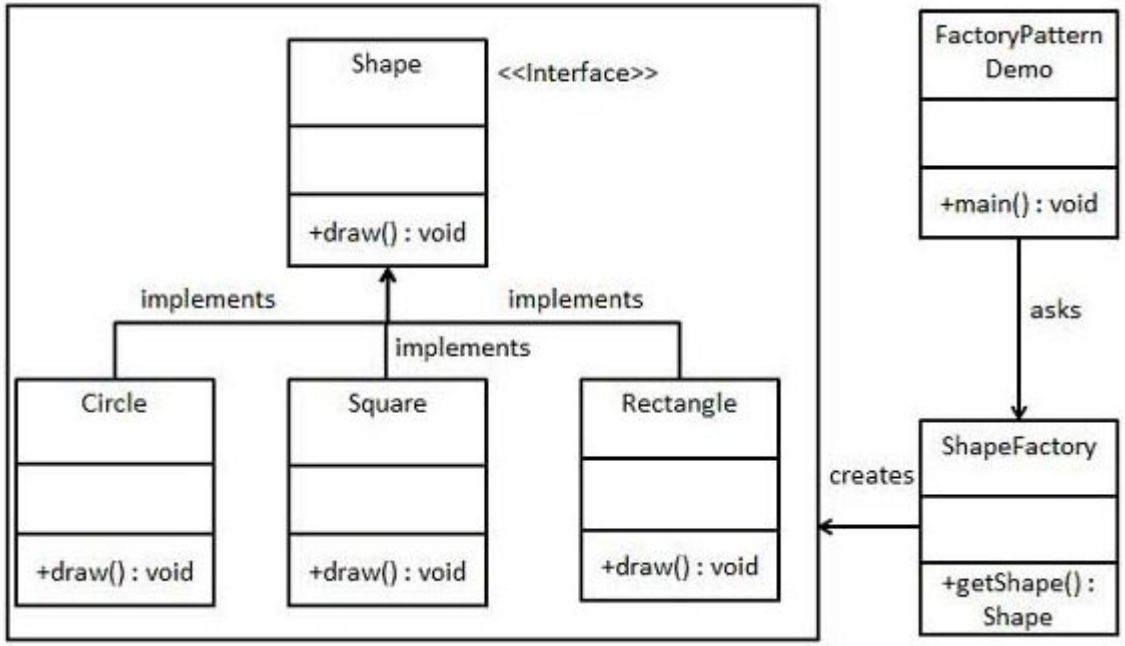
  Especificación. Parámetros que permitan a la fábrica de objetos tomar una decisión.

- **Retorno**

  Objeto específico.

Factory Method

UML



02.B

Adapter

Adapter

Adaptador

- **Objetivo**

  Proveer una interfaz común entre dos objetos que no son compatibles.

- **Uso**

  Se tiene una clase legada, que por algún motivo no puede ser modificada y debe ser integrada a nuestro sistema.
  Se crea una nueva abstracción que forzará la existencia de polimorfismo entre todas las clases.

Adapter

Adaptador

- **Requisitos**

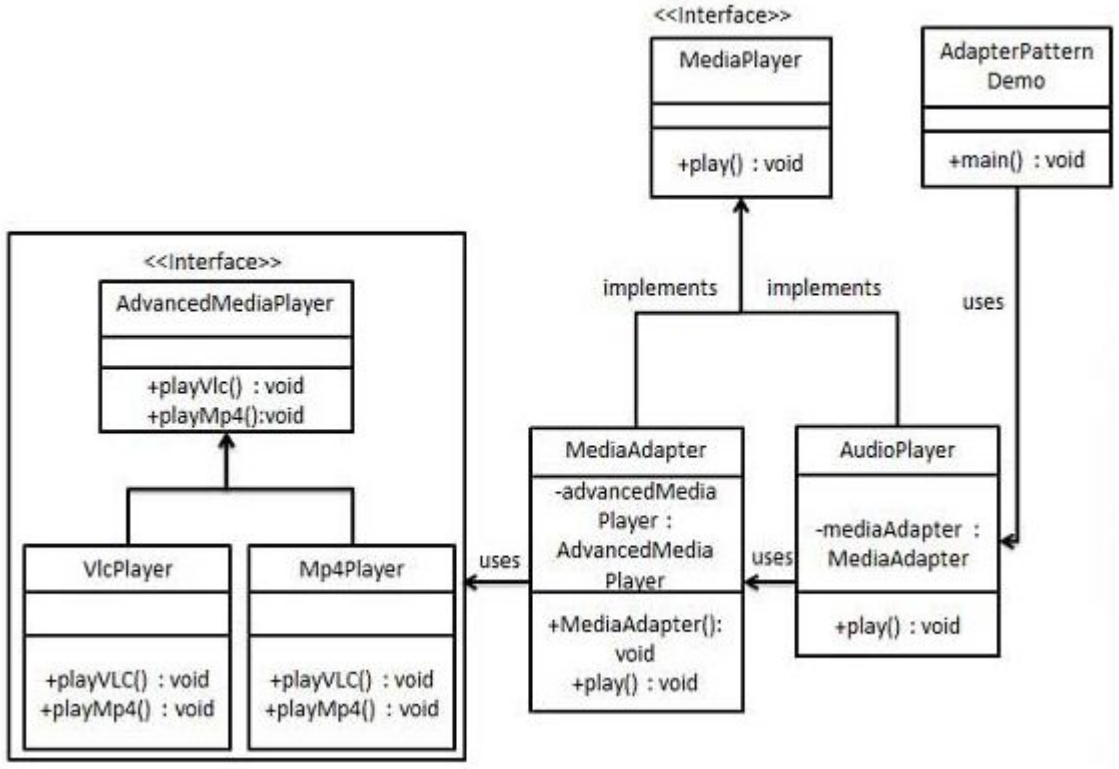
  Un **adapter** que implemente la interface que utilizará el cliente.
  Un **adaptee**, clase legada no polimorfica..

- **Retorno**

  Realiza la acción.

Adapter

UML


<<Interface>>
MediaPlayer
+play() : void
AdapterPattern Demo
+main() : void
<<Interface>>
AdvancedMediaPlayer
+playVlc() : void
+playMp4():void
implements
implements
uses
MediaAdapter
-advancedMedia Player : AdvancedMedia Player
+MediaAdapter(): void
+play() : void
AudioPlayer
-mediaAdapter : MediaAdapter
+play() : void
VlcPlayer
+playVLC() : void
+playMp4() : void
Mp4Player
+playVLC() : void
+playMp4() : void
uses
uses

02.C

State

State

Estado

- **Objetivo**

  Delegar la implementación a un estado interno del objeto.

- **Uso**

  En determinado momento y bajo ciertas circunstancias relacionadas al estado actual del objeto, la implementación de una acción debe ser.

Adapter

Estado

- **Requisitos**

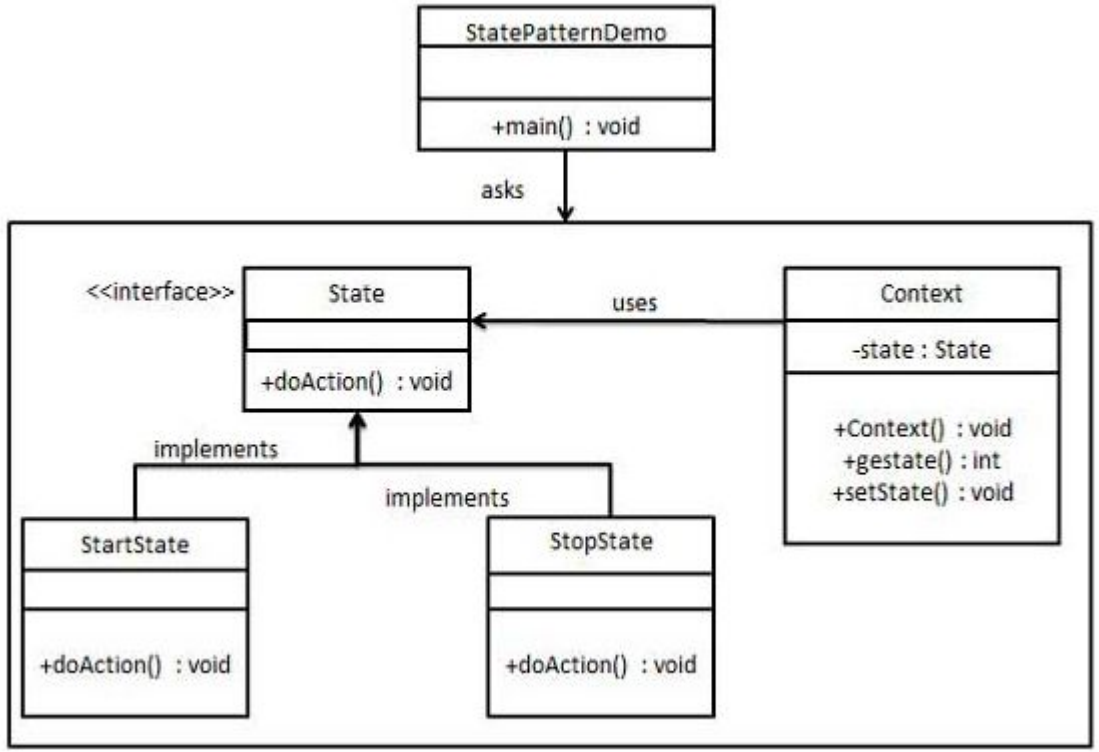
  Estados polimórficos.

- **Retorno**

  Llamada al método adecuado del objeto.

State

UML


StatePatternDemo
+main() : void
asks
<<interface>>
State
+doAction() : void
uses
Context
-state : State
+Context() : void
+gestate() : int
+setState() : void
implements
implements
StartState
+doAction() : void
StopState
+doAction() : void

02.D

Más Patrones

Referencias

- **Tutorials Point**

  https://www.tutorialspoint.com/design_pattern/index.htm

- **Refactoring Guru**

  https://refactoring.guru/design-patterns/