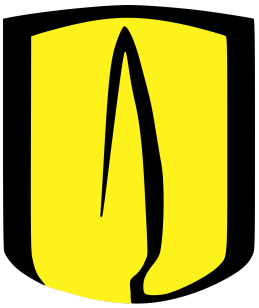
Taller 5

DPOO

Hector Duarte - 202210781

Juan David Duarte - 202215070

David Rey - 202211410



Universidad de los Andes

Bogota D.C.

Información general proyecto

El Sistema de Gestión de Propiedades (PMS) es crucial para administrar reservas, asignaciones de habitaciones y registros, seguimiento de información de huéspedes, automatización de facturación, manejo de tareas de limpieza y mantenimiento, y generación de informes para análisis. La gestión de inventario es esencial para satisfacer la demanda del cliente, guiando decisiones de compra y producción y mejorando el servicio al cliente.

El inventario incluye detalles de cada habitación, como capacidad, identificador y ubicación. Las habitaciones pueden ser suites, suites dobles o habitaciones estándar. El administrador puede agregar habitaciones individualmente o cargar datos de habitaciones desde un documento.

El sistema también gestiona una variedad de servicios del hotel, desde básicos hasta personalizados, incluyendo spa y servicios de guía turística. El sistema registra los pagos realizados y genera facturas, además de llevar un registro de los servicios consumidos por los huéspedes.

La preparación de las reservas, el registro y la facturación son aspectos esenciales para garantizar un viaje exitoso y sin estrés. Se requiere una planificación adecuada y una atención cuidadosa a las finanzas y reservas.

Estructura del patrón (Ejemplo codigo)

```
// La "abstracción" define la interfaz para la parte de
// "control" de las dos jerarquías de clase. Mantiene una
// referencia a un objeto de la jerarquía de "implementación" y
// delega todo el trabajo real a este objeto.
class RemoteControl is
    protected field device: Device
    constructor RemoteControl(device: Device) is
```

```
        this.device = device
    method togglePower() is
        if (device.isEnabled()) then
            device.disable()
        else
            device.enable()
    method volumeDown() is
        device.setVolume(device.getVolume() - 10)
    method volumeUp() is
        device.setVolume(device.getVolume() + 10)
    method channelDown() is
        device.setChannel(device.getChannel() - 1)
    method channelUp() is
        device.setChannel(device.getChannel() + 1)


// Puedes extender clases de la jerarquía de abstracción
// independientemente de las clases de dispositivo.
class AdvancedRemoteControl extends RemoteControl is
    method mute() is
        device.setVolume(0)


// La interfaz de "implementación" declara métodos comunes a
// todas las clases concretas de implementación. No tiene por
// qué coincidir con la interfaz de la abstracción. De hecho,
// las dos interfaces pueden ser completamente diferentes.
// Normalmente, la interfaz de implementación únicamente
// proporciona operaciones primitivas, mientras que la
// abstracción define operaciones de más alto nivel con base en
// las primitivas.
interface Device is
    method isEnabled()
    method enable()
    method disable()
```

```
    method getVolume()
    method setVolume(percent)
    method getChannel()
    method setChannel(channel)


// Todos los dispositivos siguen la misma interfaz.
class Tv implements Device is
    // ...

class Radio implements Device is
    // ...


// En algún lugar del código cliente.
tv = new Tv()
remote = new RemoteControl(tv)
remote.togglePower()
radio = new Radio()
remote = new AdvancedRemoteControl(radio)
```

Información general sobre el patrón: qué patrón es y para qué se usa usualmente

El patrón Bridge, también conocido como "Handle/Body", es un patrón de diseño estructural que proporciona una forma de desacoplar una abstracción de su implementación, de modo que ambas puedan variar independientemente. Esto significa que el patrón Bridge se utiliza para separar la interfaz de una clase y la implementación de esa clase en dos diferentes jerarquías de herencia.

El objetivo principal de este patrón es evitar el enlace permanente entre la abstracción y su implementación. Esto puede ser útil cuando ambas necesitan variar, cuando las

implementaciones deben ser intercambiables, o cuando tanto las abstracciones como sus implementaciones deben ser extensibles por medio de subclases.

El patrón Bridge se compone de los siguientes componentes:

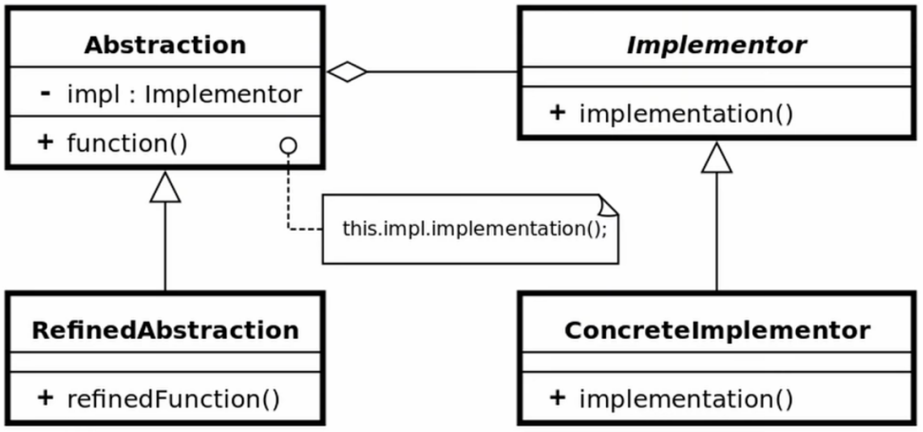
1. Abstracción Define la interfaz de la abstracción y mantiene una referencia a un objeto de tipo Implementor.

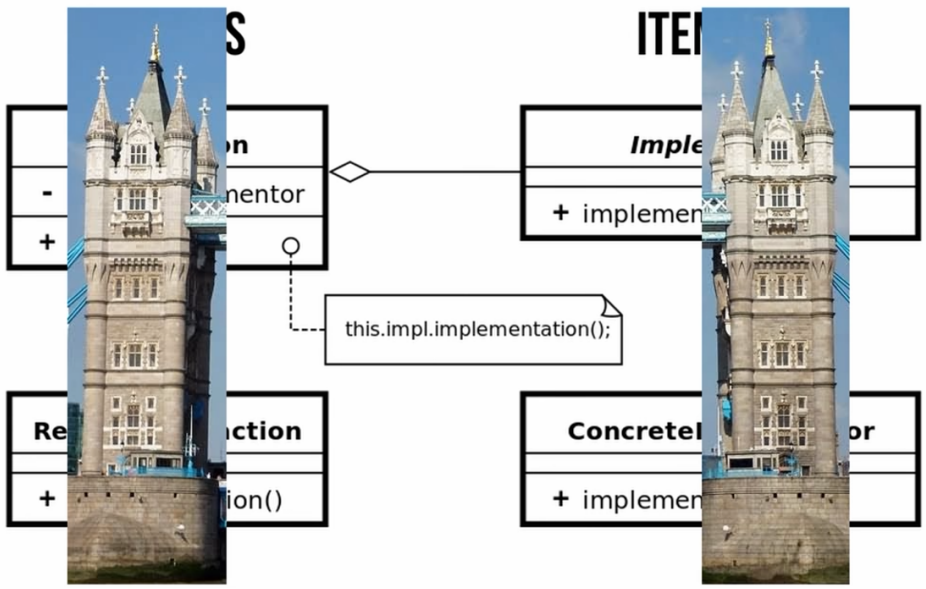
2.Implementar: Define la interfaz para las clases de implementación. Esta interfaz no tiene que corresponder exactamente a la interfaz de Abstracción y puede ser muy diferente. Típicamente la interfaz Implementor proporciona sólo operaciones primitivas, y Abstracción define operaciones de nivel superior basadas en estas operaciones primitivas.

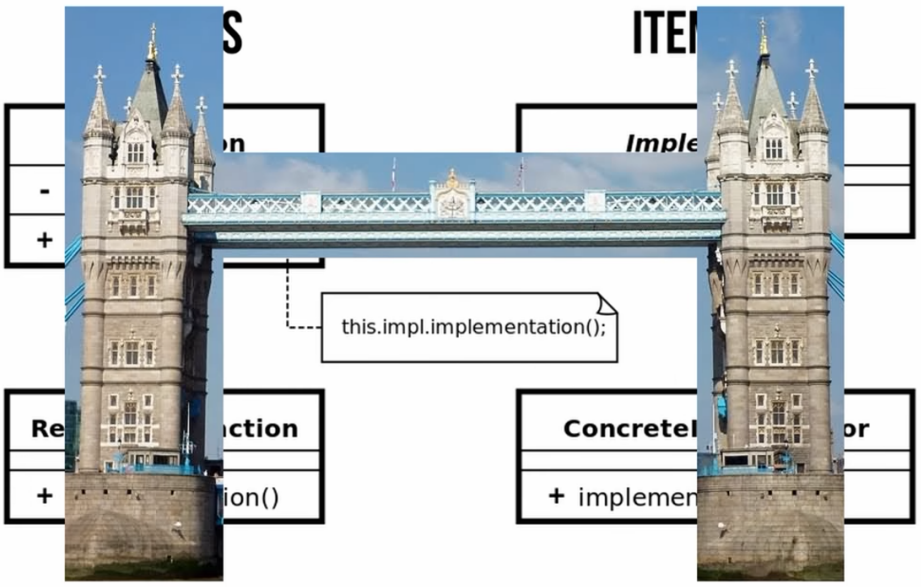
3. Refined Abstraction: Extiende la interfaz definida por Abstracción.

4. Concrete Implementor: Implementa la interfaz Implementor y define su implementación concreta.

Un ejemplo de uso de este patrón podría ser en el desarrollo de una aplicación que necesita presentar diferentes tipos de vistas de datos (como gráficos de barras, gráficos de líneas, etc.). Podríamos tener una abstracción "VistaDeDatos" y múltiples implementaciones concretas como "VistaDeDatosDeGráficoDeBarras", "VistaDeDatosDeGráficoDeLíneas", etc. De esta manera, podemos cambiar fácilmente la implementación de la vista de datos sin tener que cambiar la clase que utiliza la vista de datos.


Abstraction
- impl : Implementor
+ function()
Implementor
+ implementation()
this.impl.implementation();
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation()


this.impl.implementation();


this.impl.implementation();

Información del patrón aplicado al proyecto: explicar cómo se está utilizando el patrón dentro del proyecto

En el proyecto implementamos el patrón bridge, ya que en este contábamos con muchos servicios, por lo que resulta muchísimo más fácil utilizar este patrón a la hora de querer agregar un servicio nuevo, o removerlo en su defecto, debido a que no tenemos que cambiar y agregar muchas clases de servicios, si no que solo tenemos una clase grande llamada servicio que está contenida en el inventario, y es aquí donde se forma el puente (la contenencia). Es así que al ya tener esta conexión no tenemos que añadir clases nuevas por cada servicio, si no que solo tenemos que agregar el servicio.

La principal ventaja de usar este patrón, es que como ya lo hemos mencionado previamente, al no necesitar crear muchas clases, se facilita mucho a la hora de querer agregar o eliminar elementos, debido a que no se necesita tanto código, como si lo hiciéramos uno por uno. Además de esto, es más sencillo para el resto del grupo de trabajo entender el código de esta manera, ya que es mucho más simplificado.

Y las desventajas son principalmente que si queremos que algún servicio tenga un layout o algo especial además de los otros servicios, no podemos usar este patrón, ya que la gracia de este es que todos tengan las mismas características para poder funcionar, y que no existan errores.

Bibliografía

- Gamma, E., Helm, R. F., Johnson, R. E., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. http://www.ulb.tu-darmstadt.de/tocs/59840579.pdf
- BettaTech. [BettaTech] (2022). BRIDGE | Patrones de Diseño [Video]. Youtube.https://youtu.be/6bIHhzqMdgg
- *Bridge*. (s. f.). https://refactoring.guru/es/design-patterns/bridge