Universidad Tecnológica del Perú

Docente:

Dilian Anabel HURTADO PONCE

Tarea 11: Patrones estructurales: Patrón proxy, Patrón bridge, Patrón decorator y Patrón composite

Semana: 13

GRUPO 5

Integrantes:

Roberto Agustín Mejía Collazos

Miguel Ángel Velásquez Ysuiza

Manuel Ángel Pecho Santos

Daniel Wilfredo Sotomayor Beteta

07/11/24

Código GitHub

Image.java

ProxyImage.java

Código GitHub

RealImage.java

Bienvenido | RealImage.java | ClientImage.java | Image.java | ProxyImage.jav

src > proxy > examples > images > RealImage.java > ...

```java
Luis Burgos, hace 9 años | 1 author (Luis Burgos)
package proxy.examples.images;

Luis Burgos, hace 9 años | 1 author (Luis Burgos)
/**
 * Created by luisburgos on 21/09/15.
 */
public class RealImage implements Image {

    private String imageFileName;

    public RealImage(String imageFileName){
        this.imageFileName = imageFileName;
        loadFromDisk(imageFileName);
    }

    @Override
    public void display() {
        System.out.println("Displaying " + imageFileName);
    }

    private void loadFromDisk(String imageFileName){
        System.out.println("Loading " + imageFileName);
    }

}

```

ClientImage.java

Bienvenido | RealImage.java | ClientImage.java | Image.java | ProxyImage.java

src > proxy > examples > images > ClientImage.java

```java
Luis Burgos, hace 9 años | 1 author (Luis Burgos)
package proxy.examples.images;

/**
 * Created by luisburgos on 21/09/15.
 */
public class ClientImage  {

    Run | Debug
    public static void main(String[] args) {

        Image image = new ProxyImage(fileName:"img_10.jpg");

        //image will be loaded from disk
        image.display();
        System.out.println(x:"");

        //image will NOT be loaded from disk
        image.display();
    }

}

```

Análisis del código

Análisis del código y Uso del patrón Proxy

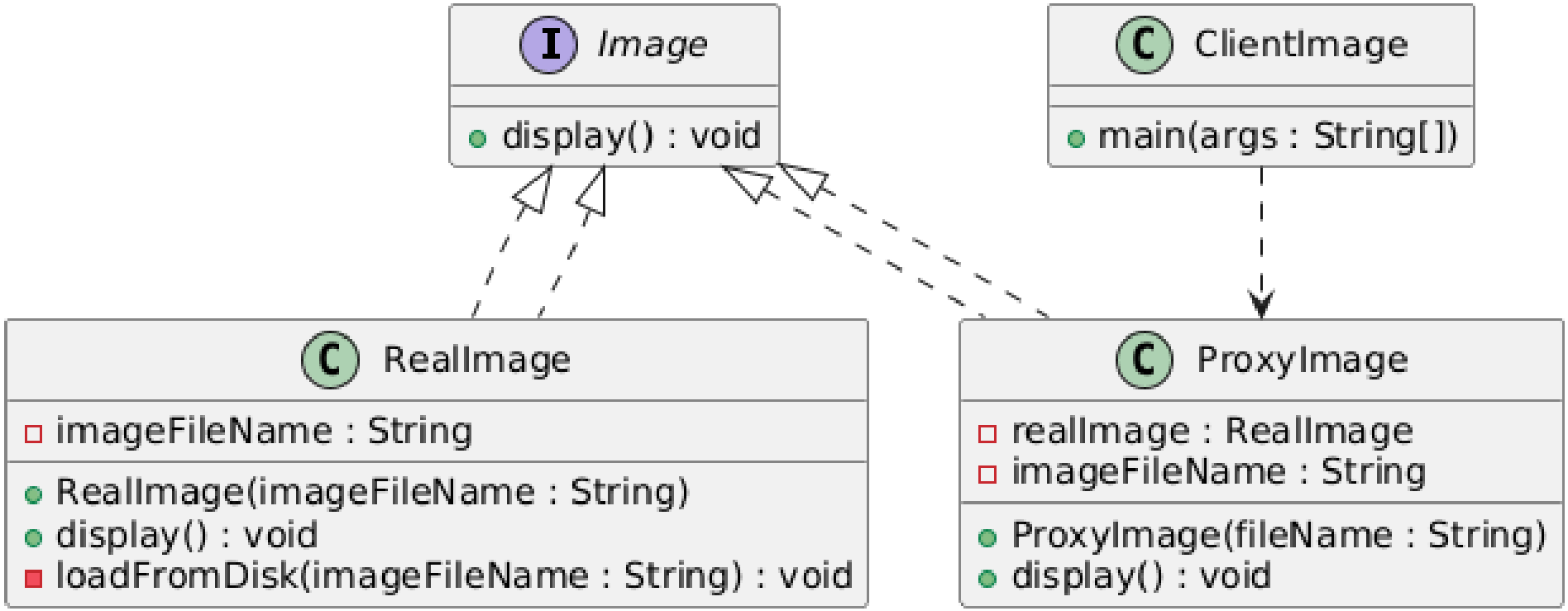
Este código implementa el patrón de diseño *Proxy*, que permite controlar el acceso a un objeto real, en este caso, la clase RealImage

1. **Interfaz Image:**
    - **Define el método display(), que permite mostrar la imagen. Esta interfaz es implementada tanto por ProxyImage como por RealImage.**
2. **Clase RealImage:**
    - **Esta es la clase que representa la imagen real. En su constructor, realiza una operación costosa de cargar la imagen desde el disco mediante el método loadFromDisk.**
    - **Al llamar a display(), muestra la imagen.**
3. **Clase ProxyImage:**
    - **Esta clase actúa como intermediaria entre el cliente y RealImage. Contiene una referencia a una RealImage y controla su creación solo cuando se llama por primera vez a display().**
    - **Esto permite que la imagen no se cargue desde el disco cada vez que se llama display(), optimizando el rendimiento.**
4. **Clase ClientImage:**
    - **Simula el uso de las imágenes a través del *Proxy*. Al llamar a display() en una ProxyImage, la imagen se carga desde el disco solo la primera vez.**

Diferencia con un Enfoque Tradicional de POO

En un enfoque tradicional, la clase ClientImage trabajaría directamente con la clase RealImage, lo que haría que la imagen se cargara desde el disco cada vez que se quisiera mostrar, independientemente de si ya se ha mostrado antes o no. En cambio, el patrón *Proxy* optimiza el uso de recursos, retrasando la carga de la imagen hasta que realmente se necesite (también conocido como *carga diferida* o *lazy loading*).

DIAGRAMA UML



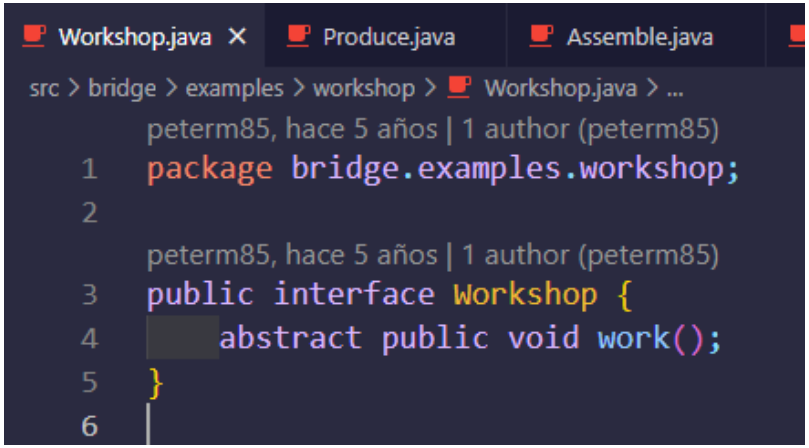
Explicación del Diagrama

Explicación del Diagrama

1. **Interfaz Image:**
    - **La interfaz Image define el método display, que es implementado por ambas clases (RealImage y ProxyImage).**
2. **Clase RealImage:**
    - **Esta clase concreta implementa la interfaz Image y representa la imagen real.**
    - **Incluye un método loadFromDisk, que se encarga de la carga costosa de la imagen.**
3. **Clase ProxyImage:**
    - **También implementa la interfaz Image, pero no carga la imagen directamente. En su lugar, retiene una referencia a RealImage y la crea solo cuando display es llamado por primera vez.**
4. **Clase ClientImage:**
    - **Es el cliente que usa el proxy ProxyImage para acceder a la image**

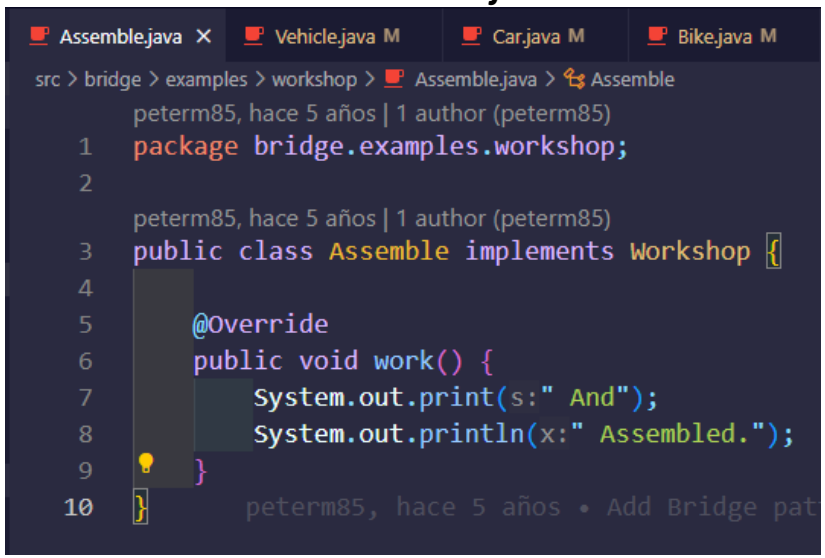
Código GitHub

Workshop.java



Produce.java

Assemble.java



Código GitHub

Vehicle.java

Assemble.java | Vehicle.java | Car.java | Bike.java | Client.java

src > bridge > examples > workshop > Vehicle.java > ...

```java
package bridge.examples.workshop;
public abstract class Vehicle {
    protected Workshop workShop1;
    protected Workshop workShop2;

    protected Vehicle(Workshop workShop1, Workshop workShop2) {
        this.workShop1 = workShop1;
        this.workShop2 = workShop2;
    }
    abstract public void manufacture();
}
```

Car.java

Car.java | Bike.java | Client.java

src > bridge > examples > workshop > Car.java > ...

```java
package bridge.examples.workshop;
public class Car extends Vehicle {
    public Car(Workshop workShop1, Workshop workShop2){
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print(s:"Car ");
        workShop1.work();
        workShop2.work();
    }
}
```

Bike.java

Bike.java | Client.java

src > bridge > examples > workshop > Bike.java > ...

```java
package bridge.examples.workshop;
public class Bike extends Vehicle {
    public Bike(Workshop workShop1, Workshop workShop2) {
        super(workShop1, workShop2);
    }

    @Override
    public void manufacture() {
        System.out.print(s:"Bike ");
        workShop1.work();
        workShop2.work();
    }
}
```

Client.java

Bike.java | Client.java

src > bridge > examples > workshop > Client.java > ...

```java
package bridge.examples.workshop;

public class Client {

    Run | Debug
    public static void main(String[] args) {
        Vehicle vehicle1 = new Car(new Produce(), new Assemble());
        vehicle1.manufacture();
        Vehicle vehicle2 = new Bike(new Produce(), new Assemble());
        vehicle2.manufacture();
    }
}
```

Análisis del código

Este código implementa el patrón de diseño Bridge, que permite separar una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.

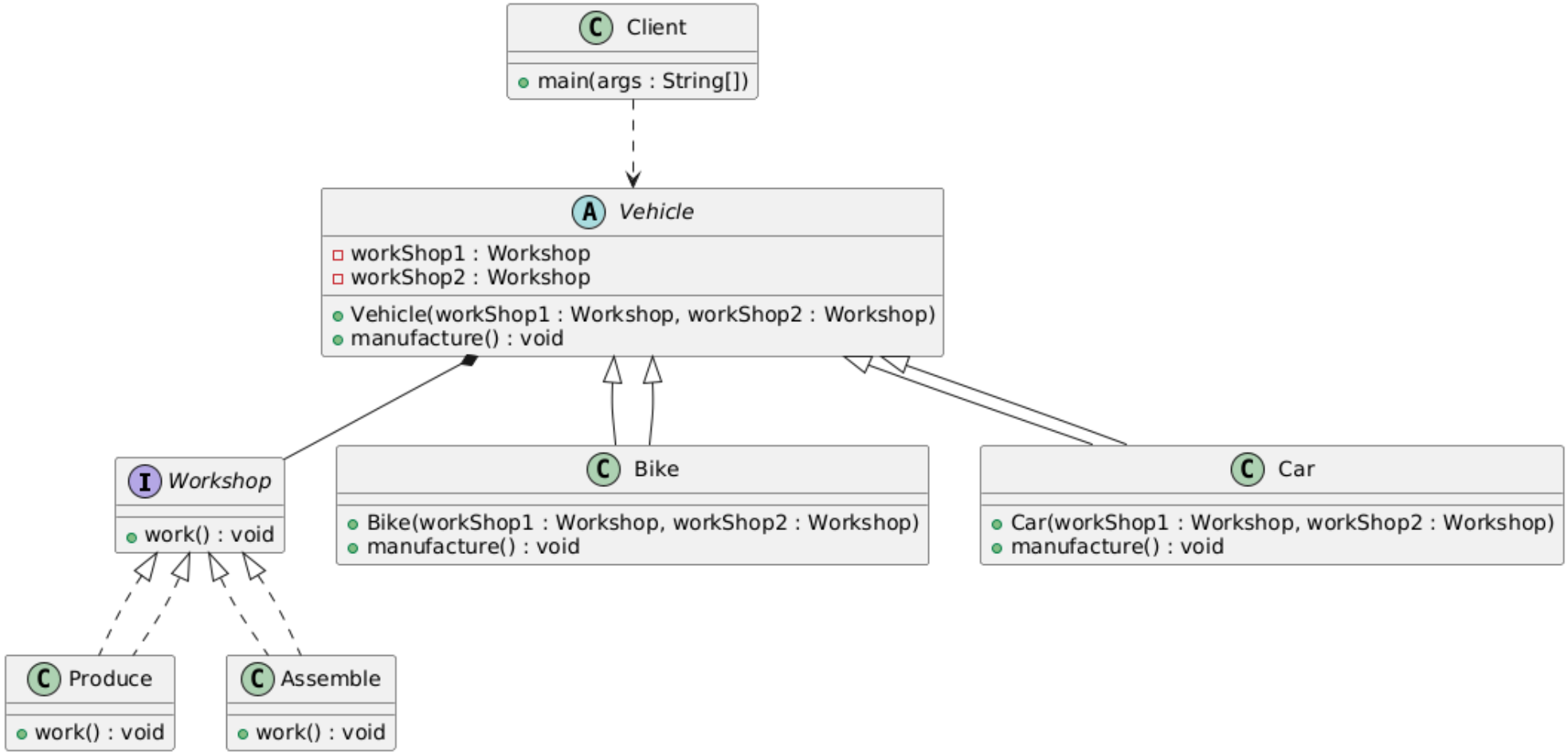
Análisis del código

1. ***Interfaz Workshop:***
    - ***Define el método work() que representa el trabajo de un taller.***
2. ***Clases Produce y Assemble:***
    - ***Ambas implementan la interfaz Workshop, cada una representando un tipo de trabajo: Produce produce un producto, y Assemble ensambla el producto.***
3. ***Clase abstracta Vehicle:***
    - ***Es la abstracción principal. Contiene dos referencias a Workshop, workShop1 y workShop2, lo que permite que cada Vehicle trabaje con diferentes talleres.***
    - ***Define el método abstracto manufacture(), que cada vehículo concreto debe implementar.***
4. ***Clases Bike y Car:***
    - ***Estas clases concretas heredan de Vehicle y representan tipos específicos de vehículos.***
    - ***En el método manufacture(), ejecutan el trabajo de ambos talleres asociados.***
5. ***Clase Client:***
    - ***Crea instancias de Car y Bike, cada una con los talleres Produce y Assemble, y luego llama al método manufacture() para realizar el trabajo.***

Diferencia con un Enfoque Tradicional de POO

En un enfoque tradicional, la relación entre Vehicle y Workshop estaría más rígidamente estructurada, haciendo difícil cambiar o extender los tipos de trabajo (talleres) o vehículos sin modificar la jerarquía existente. Con el patrón Bridge, las clases Vehicle y Workshop pueden variar independientemente: se pueden añadir nuevos tipos de vehículos o talleres sin afectar directamente a las otras clases.

DIAGRAMA UML



Explicación UML

Explicación del Diagrama UML

1. **Interfaz Workshop**:
    - Define el método work() y es implementada por Produce y Assemble, permitiendo que ambas clases realicen el trabajo específico de cada taller.
2. **Clase Vehicle**:
    - Es una abstracción que contiene dos referencias a Workshop (workShop1 y workShop2), lo que permite que cada instancia de Vehicle use diferentes talleres.
    - El método manufacture() es abstracto, por lo que debe ser implementado por las subclases concretas.
3. **Clases Bike y Car**:
    - Representan tipos específicos de Vehicle que implementan el método manufacture() y usan los talleres especificados (Produce y Assemble en este caso).
4. **Clase Client**:
    - Es el cliente que crea y usa las instancias de Vehicle (en este caso, Car y Bike) con las instancias de Workshop.