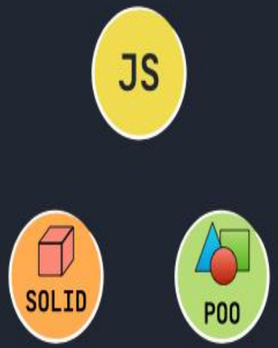
Patrones de diseño

Sesión 02:

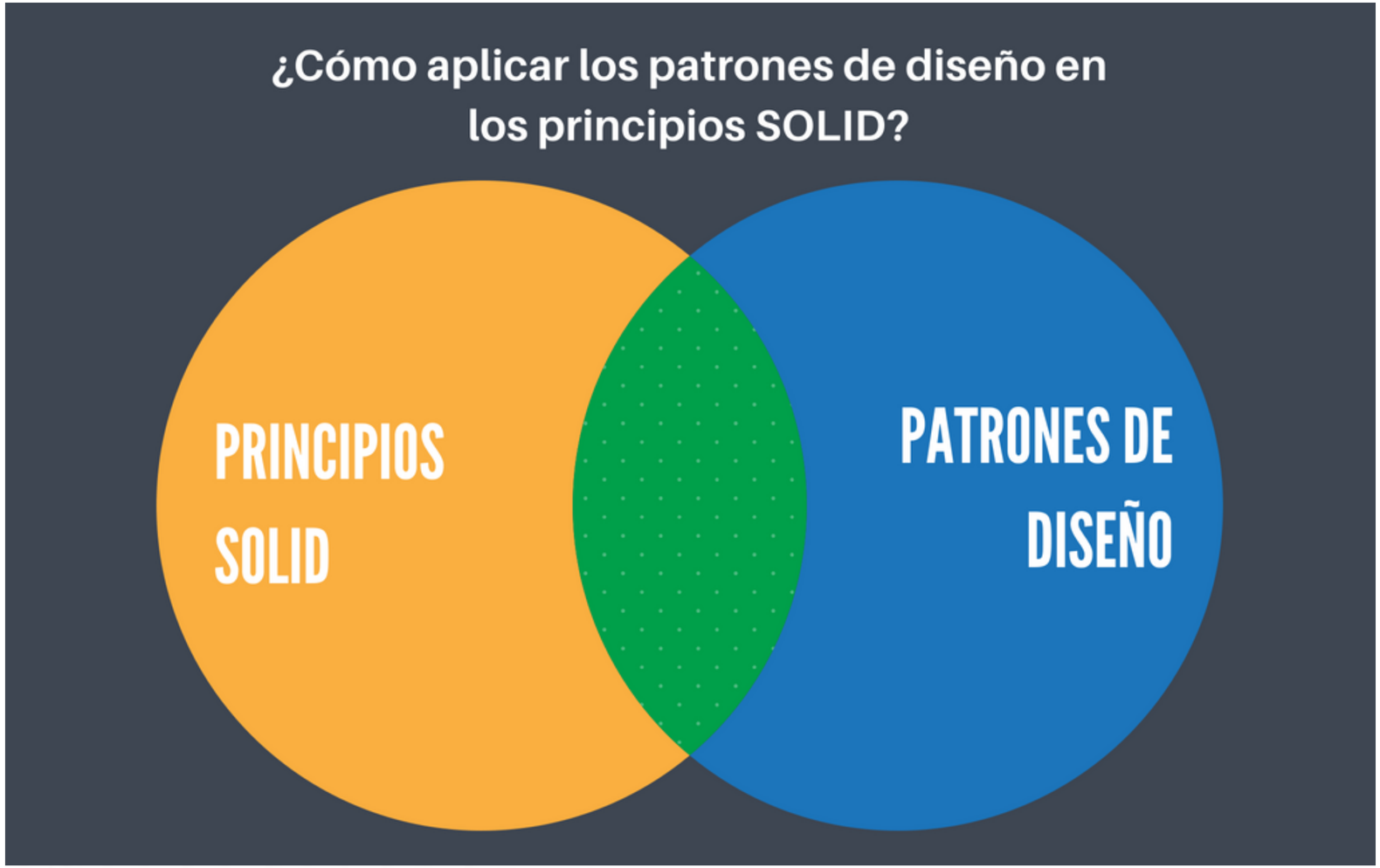
PRINCIPIOS SOLID: PRINCIPIO DE RESPONSABILIDAD UNICA, SUSTITUCIÓN DE LISKOV y PRINCIPIO ABIERTO/CERRADO

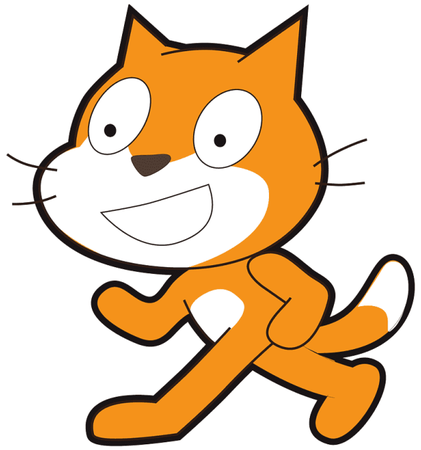
Principios de patrones de diseño

```
class FinalArticle extends ArticlePart1(ArticlePart2) {

   constructor() {
        super();
        this.title = “Los cinco principios de SOLID";
       this.version = “Final”;
       this.subTitle = “Aplicados en JavaScript”;
   }

   get message() {
        const { version, title, subTitle } = this;
        return `${version} ${title}, ${subTitle}`;
   }
}

const ArticleTheme = new ArticlePart3();
console.log(ArticleTheme.message);

// Final: Los cinco principios de SOLID, Aplicados en JavaScript
```



Universidad Tecnológica del Perú





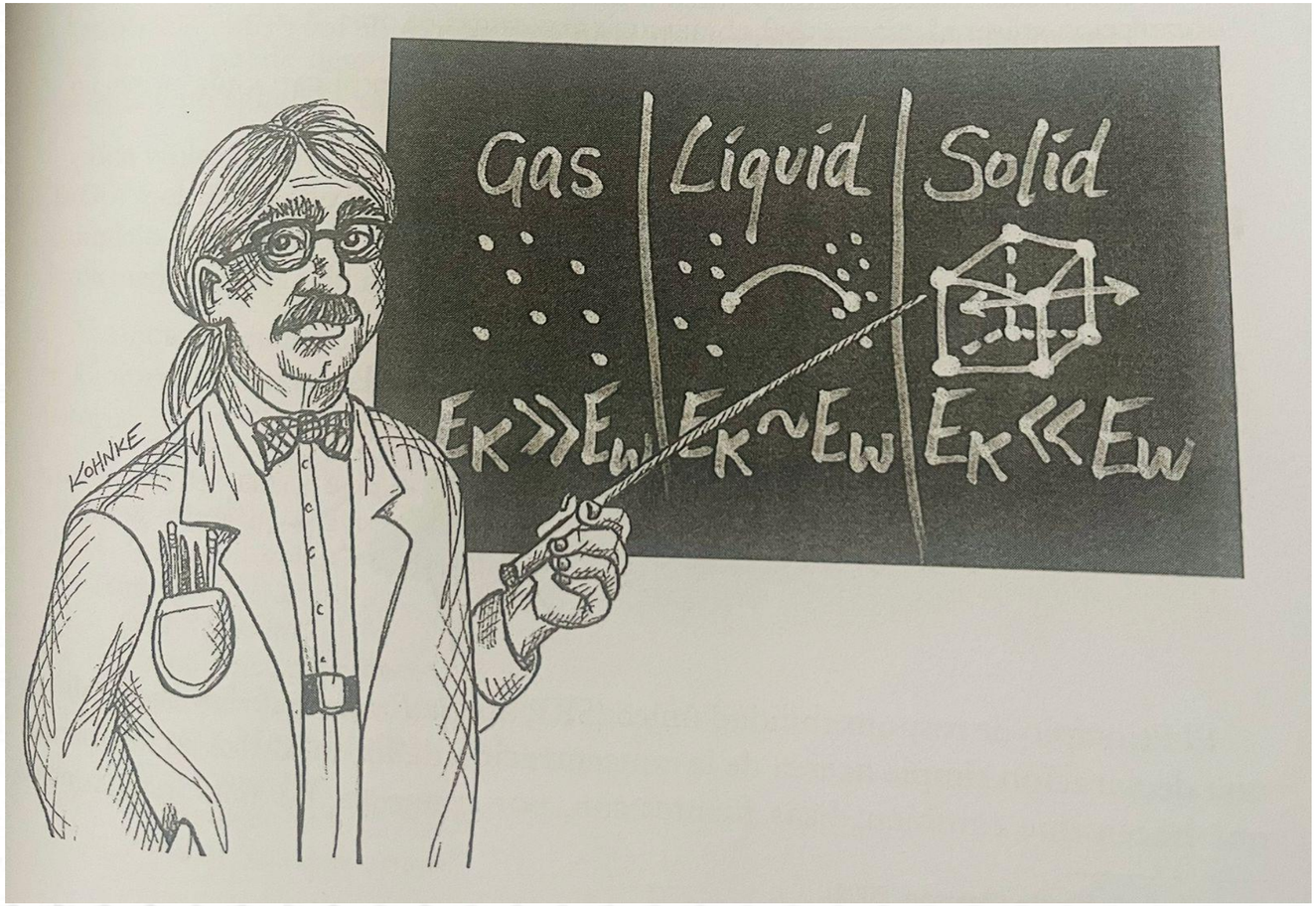
Universidad Tecnológica del Perú

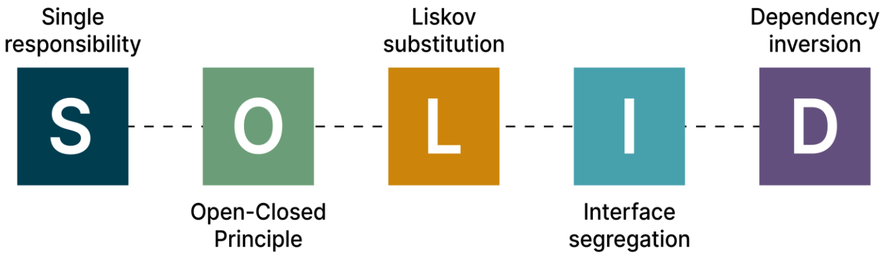
Logro de la sesión

Al finalizar la sesión, el estudiante elabora su programación usando los principios SOLID: principio de responsabilidad única, sustitución de LISKOV y principio abierto/cerrado empleando los diversos patrones de diseño aplicados a casos matemáticos, financieros y físicos.

Universidad Tecnológica del Perú

Principios SOLID





Universidad Tecnológica del Perú

¿Qué son los principios SOLID?

Los principios SOLID hace más de dos décadas en el contexto del diseño OO. Debido a ese contexto, muchos han llegado a asociar esos principios con la OO y los consideran un **anatema** para la programación funcional.

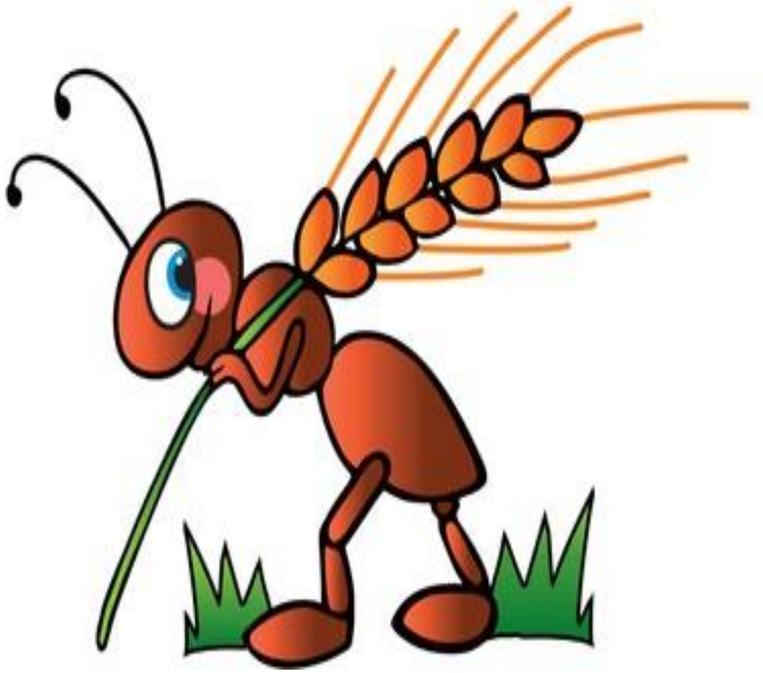
Es desafortunado porque los principios SOLID son principios generales de diseño de software que no son específicos de ningún estilo de programación concreto.

Explicar cómo se aplican los principios SOLID a la programación funcional.

Universidad Tecnológica del Perú

Principio de responsabilidad única (SRP)

El Principio de responsabilidad única **(SRP, Single Responsibility Principle)** es una declaración simple acerca de la concentración de los módulos en las fuentes que hacen que cambien. Esas fuentes son, por supuesto, las personas.



Son las personas quienes solicitan **cambios en el software y, por tanto, las personas son las responsables de nuestros módulos.** Estas personas pueden separarse en grupos llamados roles o actores. Un actor es una persona, o un grupo de personas, que requiere las mismas cosas del sistema. Los tipos de cambios que solicitan son coherentes entre sí. Por otra parte, actores diferentes tienen necesidades distintas.

UTP Universidad Tecnológica del Perú

Los cambios que solicite un actor afectarán al sistema de maneras muy diferentes a los cambios solicitados por otros actores. Esos cambios dispares pueden incluso tener propósitos contradictorios.

Cuando un módulo es responsable ante más de un actor, los cambios solicitados por actores competidores pueden interferir entre sí. A menudo, esta interferencia lleva a un olor de diseño de fragilidad; hace que el sistema se estropee de maneras inesperadas cuando se realizan cambios simples.

Nada es tan aterrador para directores y clientes como un sistema que de pronto se comporta mal de un modo alarmante tras realizar cambios simples en las características. Si esto se repite con demasiada frecuencia, la única conclusión a la que se puede llegar es que los desarrolladores han perdido el control del sistema y no saben lo que están haciendo.

Una violación del SRP puede ser algo tan simple como mezclar el formato de la GUI y el código de las reglas de negocio en el mismo módulo, o puede ser tan complejo como utilizar procedimientos almacenados en la base de datos para implementar reglas de negocio.

Como su propio nombre indica, establece que una clase, componente o microservicio debe ser responsable de una sola cosa (el tan aclamado término “decoupled” en inglés).

Si por el contrario, una clase tiene varias responsabilidades, esto implica que el cambio en una responsabilidad provocará la modificación en otra responsabilidad.

Considera este ejemplo:

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }

    void guardarCocheDB(Coche coche){ ... }
}
```

Universidad Tecnológica del Perú

¿Por qué este código viola el principio de responsabilidad única?

Como podemos observar, la clase Coche permite tanto el **acceso** a las propiedades de la clase como a realizar operaciones sobre la BBDD, por lo que **la clase ya tiene más de una responsabilidad.**

Supongamos que debemos realizar cambios en los métodos que realizan las operaciones a la BBDD.

En este caso, además de estos cambios, probablemente tendríamos que tocar los nombres o tipos de las propiedades, métodos, etc, cosa que no parece muy eficiente porque solo estamos modificando cosas que tienen que ver con la BBDD, *¿verdad?*

Para evitar esto, debemos separar las responsabilidades de la clase, por lo que podemos crear otra clase que se encargue de las operaciones a la BBDD:

Universidad Tecnológica del Perú

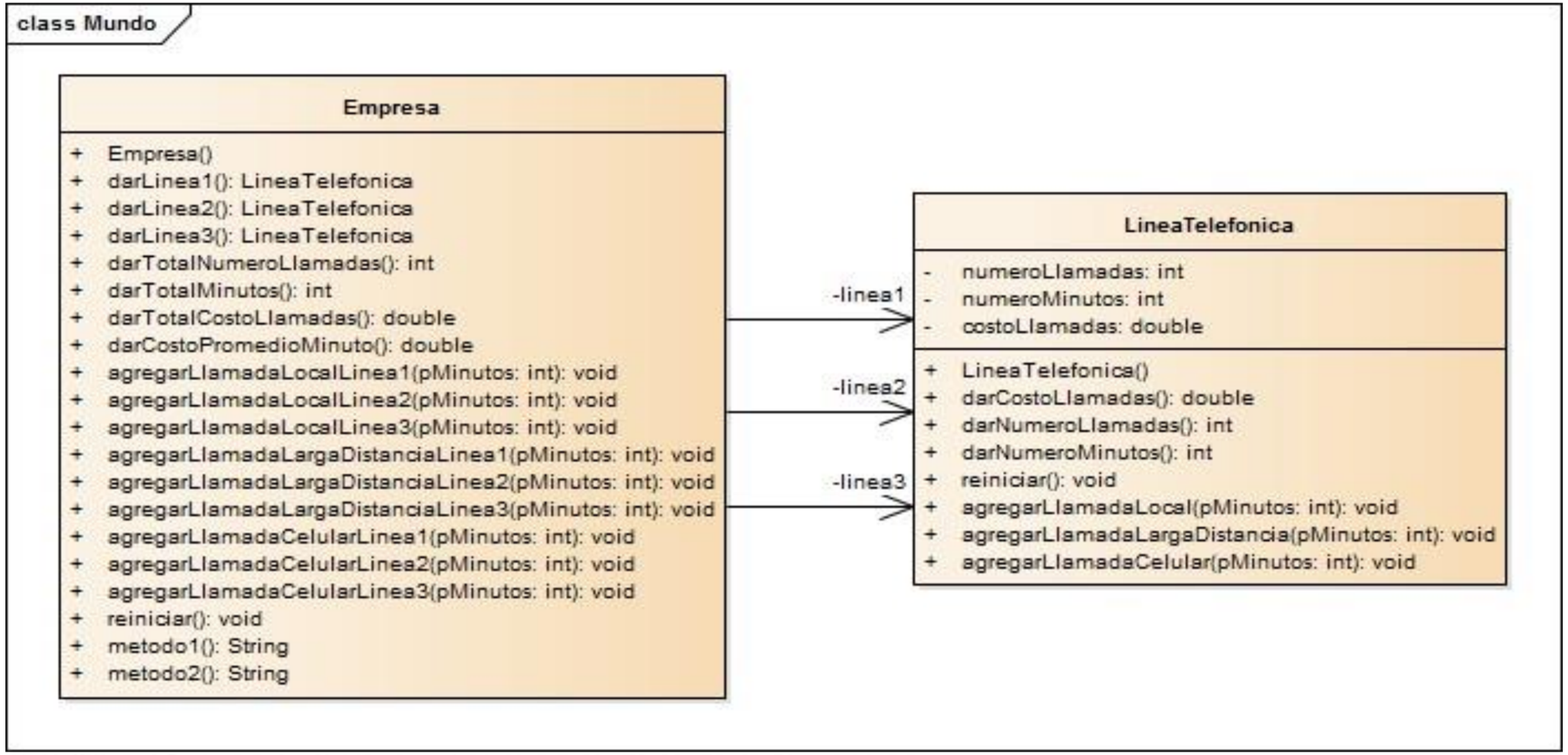
Nuestro programa será mucho más cohesivo y estará más encapsulado aplicando este principio.

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}

class CocheDB{
    void guardarCocheDB(Coche coche){ ... }
    void eliminarCocheDB(Coche coche){ ... }
}
```

WAOOO!!!

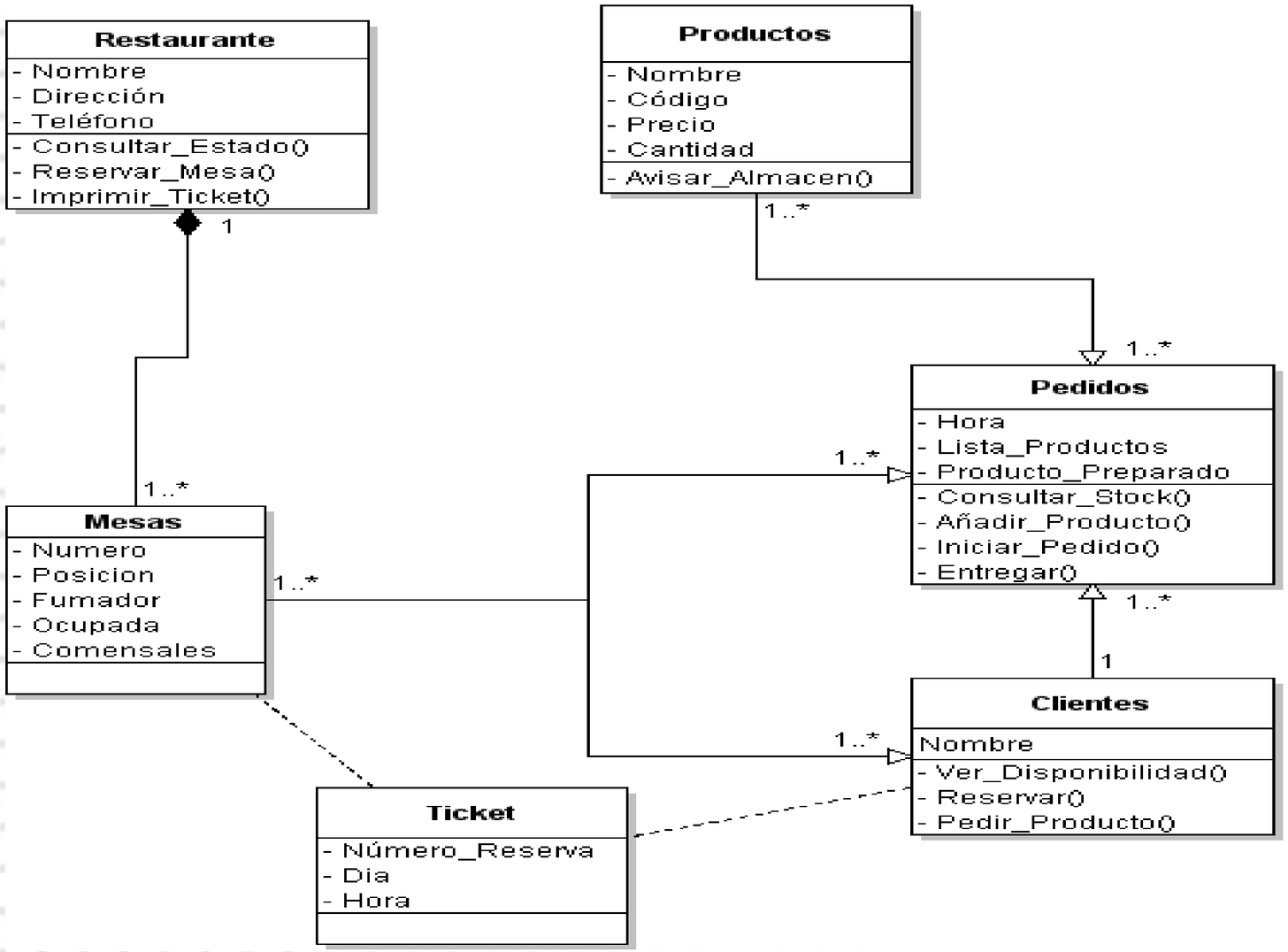
Universidad Tecnológica del Perú

Ejercicio1



Universidad Tecnológica del Perú

Ejercicio2



Universidad Tecnológica del Perú

Principio de Abierto – Cerrado (OCP)

El Principio de abierto-cerrado (OCP, Open-Closed Principle) lo enunció por primera vez Bertrand Meyer en su libro clásico de 1988,

Construcción de software orientado a objetos.

Parafraseando, dice que los módulos de software deberían estar abiertos a la ampliación, pero cerrados a la modificación.

Eso significa que nos conviene diseñar los módulos de manera que ampliar o cambiar su comportamiento no requiera modificar su código.

Universidad Tecnológica del Perú

Principio de Abierto – Cerrado (OCP)

Establece que las entidades software (clases, módulos y funciones) deberían estar abiertos para su extensión, pero cerrados para su modificación.

Si seguimos con la clase Coche:

```
class Coche {
    String marca;

    Coche(String marca){ this.marca = marca; }

    String getMarcaCoche(){ return marca; }
}
```

Universidad Tecnológica del Perú

Si quisiéramos iterar a través de una lista de coches e imprimir sus marcas por pantalla:

```
public static void main(String[] args) {
    Coche[] arrayCoches = {
            new Coche("Renault"),
            new Coche("Audi")
    };
    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
    }
}
```

Esto no cumpliría el principio abierto/cerrado, ya que si decidimos añadir un nuevo coche de otra marca:

```
Coche[] arrayCoches = {
        new Coche("Renault"),
        new Coche("Audi"),
        new Coche("Mercedes")
};
```

También tendríamos que modificar el método que hemos creado anteriormente:

```
public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        if(coche.marca.equals("Renault")) System.out.println(18000);
        if(coche.marca.equals("Audi")) System.out.println(25000);
        if(coche.marca.equals("Mercedes")) System.out.println(27000);
    }
}
```

```java
abstract class Coche {
    // ...
    abstract int precioMedioCoche();
}

class Renault extends Coche {
    @Override
    int precioMedioCoche() { return 18000; }
}

class Audi extends Coche {
    @Override
    int precioMedioCoche() { return 25000; }
}

class Mercedes extends Coche {
    @Override
    int precioMedioCoche() { return 27000; }
}

public static void main(String[] args) {

    Coche[] arrayCoches = {
            new Renault(),
            new Audi(),
            new Mercedes()
    };

    imprimirPrecioMedioCoche(arrayCoches);
}

public static void imprimirPrecioMedioCoche(Coche[] arrayCoches){
    for (Coche coche : arrayCoches) {
        System.out.println(coche.precioMedioCoche());
    }
}
```

Como podemos ver, para cada nuevo coche habría que añadir nueva lógica al método **precioMedioCoche().**

Esto es un ejemplo sencillo, pero imagina que tu aplicación crece y crece… ¿cuántas modificaciones tendríamos que hacer?

Mejor evitarnos esta pérdida de tiempo y dolor de cabeza, ¿verdad?

Para que cumpla con este principio podríamos hacer lo siguiente:

Conclusión:

Cada coche extiende la clase abstracta Coche e implementa el método abstracto **precioMedioCoche().**

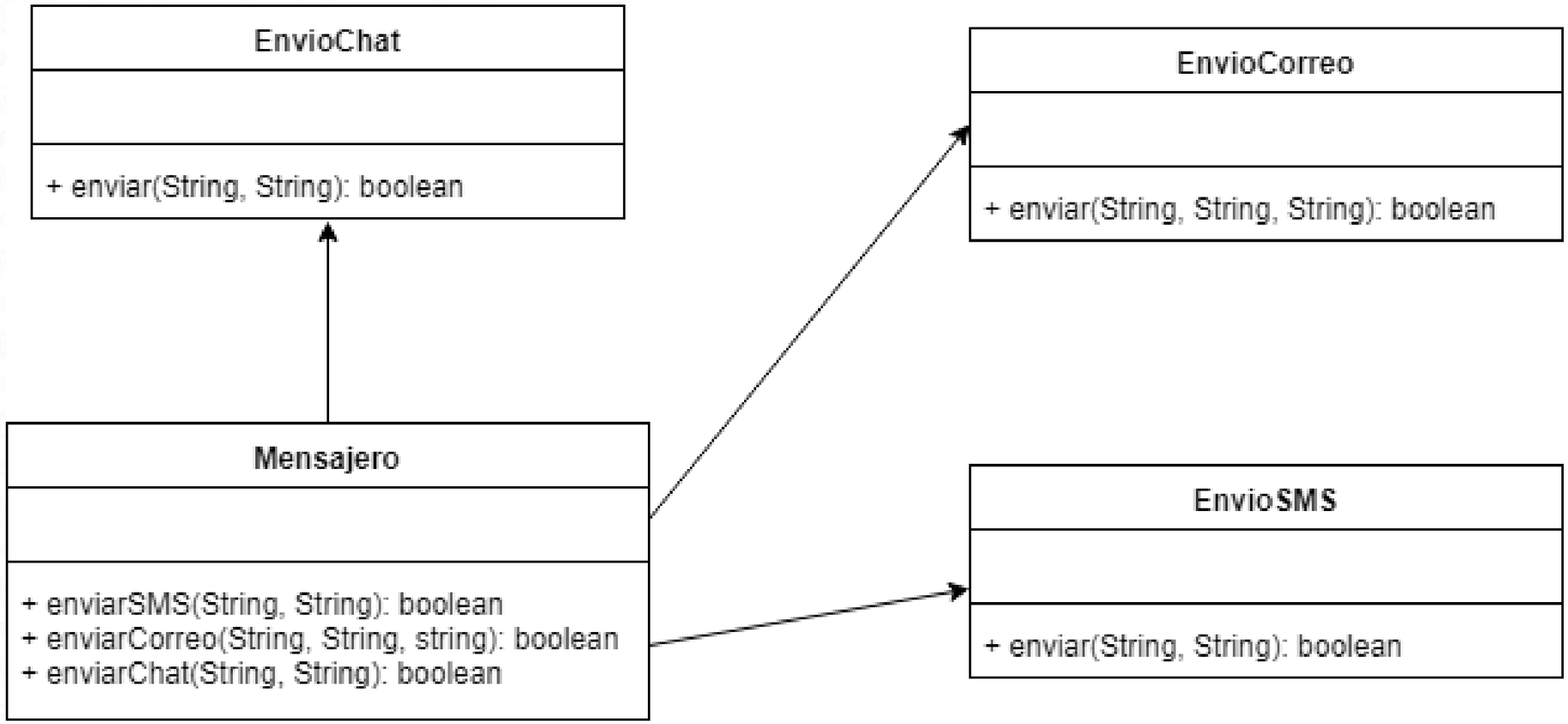
Así, cada coche tiene su propia implementación del método **precioMedioCoche(),** por lo que el método **imprimirPrecioMedioCoche()** itera el array de coches y solo llama al método **precioMedioCoche().**

Ahora, si añadimos un nuevo coche, **precioMedioCoche()** no tendrá que ser modificado. Solo tendremos que añadir el nuevo coche al array, cumpliendo así el principio abierto/cerrado.

```
public abstract class classAstratta {
    public abstract void AbstractMethod1();
        //Metodo astratto di tipo Void
    public abstract String AbstractMethod2();
        //Metodo astratto di tipo String
    public abstract int AbstractMethod3();
        //Metodo astratto di tipo Int
    public abstract double AbstractMethod4();
        //Metodo astratto di tipo Double
}
```

Universidad Tecnológica del Perú

Ejercicios



Universidad Tecnológica del Perú

Principio de sustitución de Liskov (LSP)

Cualquier lenguaje compatible con el OCP debe soportar también el Principio de sustitución de Liskov (LSP, Liskov Substitution Principle). Los dos principios están ligados porque cualquier violación del LSP es una violación latente del OCP.

El LSP lo describió por primera vez Barbara Liskov en 1988, proporcionando una definición más o menos formal de subtipo. En esencia, dijo que un subtipo debe ser sustituible por su tipo base en cualquier programa que emplee el tipo base.

Declara que una subclase debe ser sustituible por su superclase, y si al hacer esto, el programa falla, estaremos violando este principio.

Cumpliendo con este principio se confirmará que nuestro programa tiene una jerarquía de clases fácil de entender y un código reutilizable.

Veamos un ejemplo:

¿Cómo detectamos que no cumplimos con el LSP?

La forma más sencilla de detectar el incumplimiento del LSP es observando las clases que extienden de otras clases. Si estas poseen métodos sobrescritos que devuelven null o lanzan excepciones sabiendas de que la clase de la que heredan no lo hace, estamos ante una clara violación del LSP.

Si por otra parte estamos heredando de clases abstractas que nos obligan también a lanzar excepciones o retornar null, también estamos ante un caso de violación del LSP.

```
// ...
public static void imprimirNumAsientos(Coche[] arrayCoches){
     for (Coche coche : arrayCoches) {
          if(coche instanceof Renault)
               System.out.println(numAsientosRenault(coche));
          if(coche instanceof Audi)
               System.out.println(numAsientosAudi(coche));
          if(coche instanceof Mercedes)
               System.out.println(numAsientosMercedes(coche));
     }
}
imprimirNumAsientos(arrayCoches);
```

Esto viola tanto el principio de substitución de Liskov como el de abierto/cerrado.
Así, si añadimos un nuevo coche, el método debe modificarse para aceptarlo.
El programa debe conocer cada tipo de Coche y llamar a su método numAsientos() asociado.

```
// ...
Coche[] arrayCoches = {
          new Renault(),
          new Audi(),
          new Mercedes(),
          new Ford()
};

public static void imprimirNumAsientos(Coche[] arrayCoches){
     for (Coche coche : arrayCoches) {
          if(coche instanceof Renault)
               System.out.println(numAsientosRenault(coche));
          if(coche instanceof Audi)
               System.out.println(numAsientosAudi(coche));
          if(coche instanceof Mercedes)
               System.out.println(numAsientosMercedes(coche));
          if(coche instanceof Ford)
               System.out.println(numAsientosFord(coche));
     }
}
imprimirNumAsientos(arrayCoches);
```

Para que este método cumpla con el principio, seguiremos estos principios:

Si la superclase (Coche) tiene un método que acepta un parámetro del tipo de la superclase (Coche), entonces su subclase (Renault) debería aceptar como argumento un tipo de la superclase (Coche) o un tipo de la subclase (Renault).
Si la superclase devuelve un tipo de ella misma (Coche), entonces su subclase (Renault) debería devolver un tipo de la superclase (Coche) o un tipo de la subclase (Renault).
Si volvemos a implementar el método anterior:

```
public static void imprimirNumAsientos(Coche[] arrayCoches){
        for (Coche coche : arrayCoches) {
            System.out.println(coche.numAsientos());
        }
    }

imprimirNumAsientos(arrayCoches);
```

Universidad Tecnológica del Perú

Ahora al método no le importa el tipo de la clase, simplemente llama al método numAsientos() de la superclase. Solo sabe que el parámetro es de tipo coche, ya sea Coche o alguna de las subclases.

Para esto, ahora la clase Coche debe definir el nuevo método:

```
abstract class Coche {

    // ...
    abstract int numAsientos();
}
```

Y las subclases deben implementar dicho método:

```
class Renault extends Coche {

    // ...
    @Override
    int numAsientos() {
        return 5;
    }
}
// ...
```

Como podemos ver, ahora el método **imprimirNumAsientos()** no necesita saber con qué tipo de coche va a realizar su lógica, simplemente llama al método numAsientos() del tipo Coche, ya que por contrato, una subclase de Coche debe implementar dicho método.

Errores Comunes

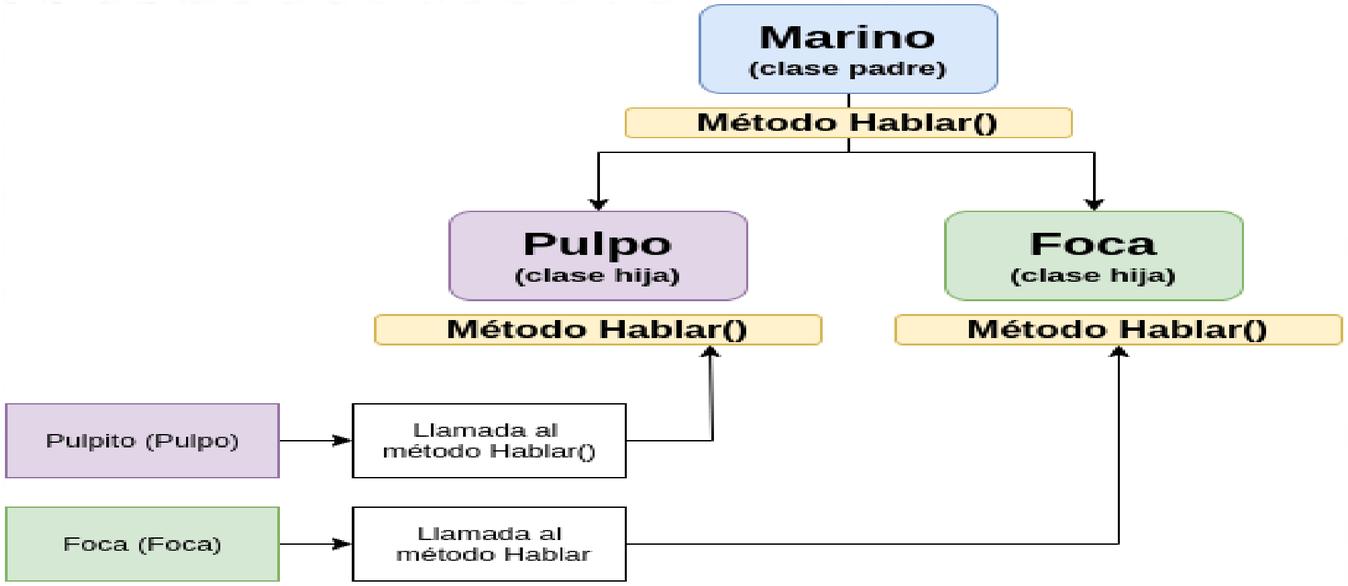
Al implementar el Principio de Sustitución de Liskov, es fácil cometer errores. Aquí hay algunos errores comunes y cómo evitarlos:

1. No Respetar el comportamiento de la Clase Base

Si una subclase altera el comportamiento esperado de la clase base, puede romper el LSP. Por ejemplo, si una subclase lanza una excepción que la clase base no lanza, esto puede causar problemas.

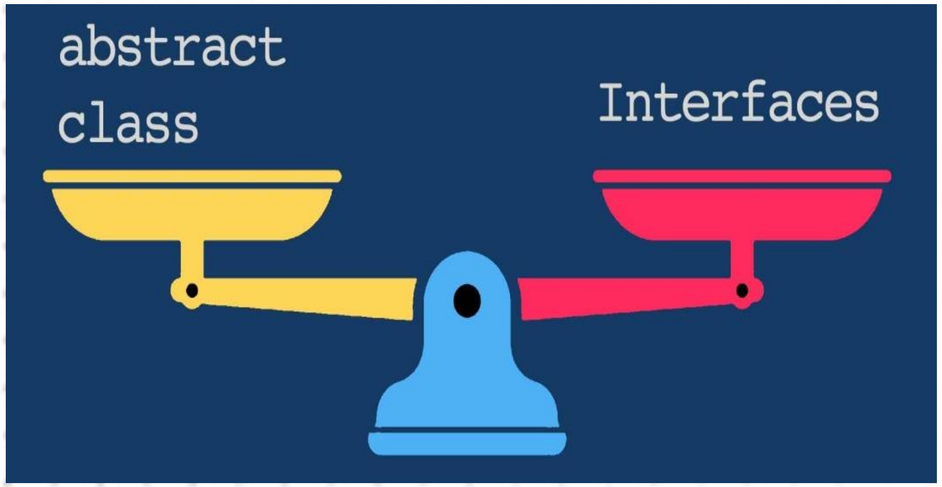
2. No Mantener la Consistencia de los Tipos

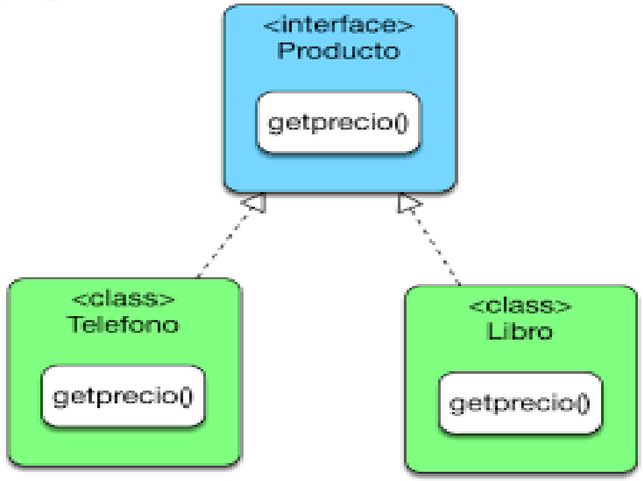
Las subclases deben ser capaces de manejar todas las entradas que la clase base puede manejar. Si una subclase no puede manejar ciertos tipos de datos que la clase base puede, esto viola el LSP.



Mejores Prácticas

- Usar interfaces para definir contratos claros.
- Evitar el uso de tipos específicos en métodos de subclases.
- Probar exhaustivamente las subclases para asegurarse de que cumplen con el comportamiento de la clase base.





Universidad Tecnológica del Perú

1. Uso de Genéricos

```java
public class Caja {
    private T contenido;

    public void setContenido(T contenido) {
        this.contenido = contenido;
    }

    public T getContenido() {
        return contenido;
    }
}
```

En este ejemplo, la clase Caja puede contener cualquier tipo de objeto, lo que permite una mayor flexibilidad y adherencia al LSP.

Universidad Tecnológica del Perú

2. Patrones de Diseño

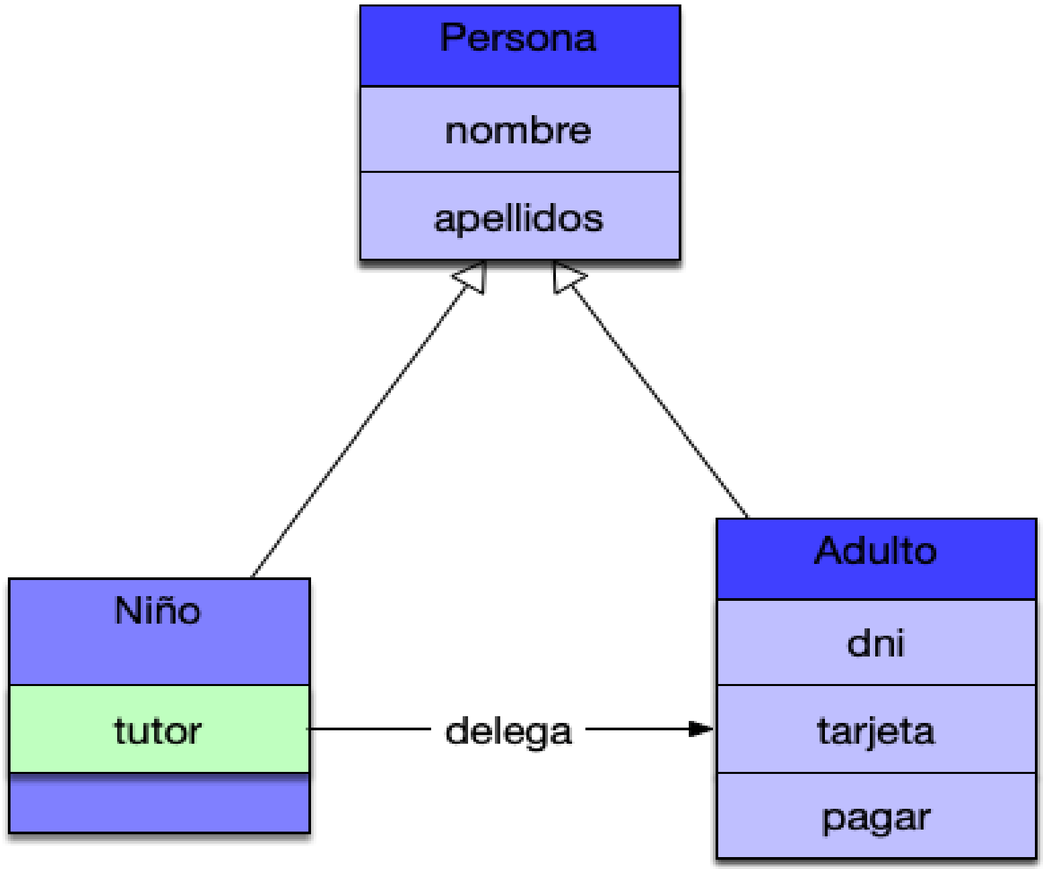
El patrón de diseño Strategy es un buen ejemplo de cómo aplicar el LSP. Este patrón permite que una familia de algoritmos se defina y se encapsule en clases separadas, permitiendo que los algoritmos sean intercambiables.

```java
public interface Estrategia {
    void ejecutar();
}

public class EstrategiaA implements Estrategia {
    @Override
    public void ejecutar() {
        System.out.println("Estrategia A ejecutada");
    }
}

public class EstrategiaB implements Estrategia {
    @Override
    public void ejecutar() {
        System.out.println("Estrategia B ejecutada");
    }
}
```

En este caso, las clases Estrategia A y Estrategia B pueden ser usadas indistintamente sin romper el comportamiento del programa.

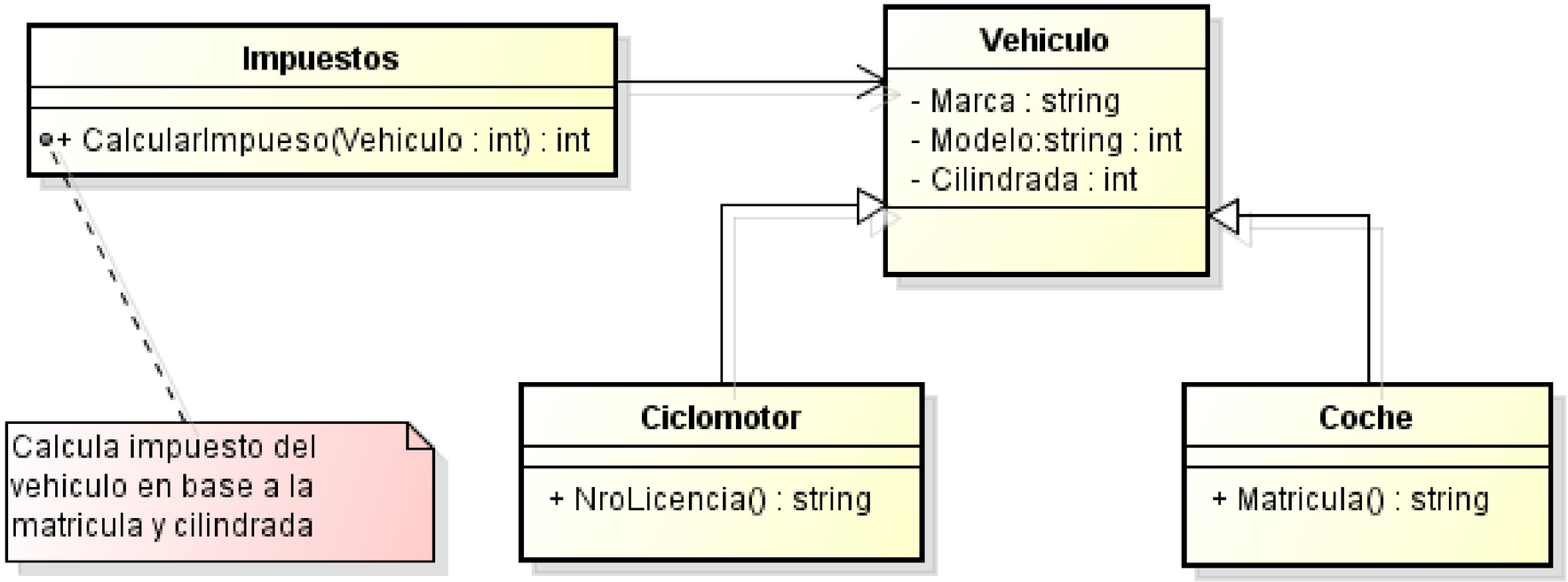
Universidad Tecnológica del Perú

Ejercicio1



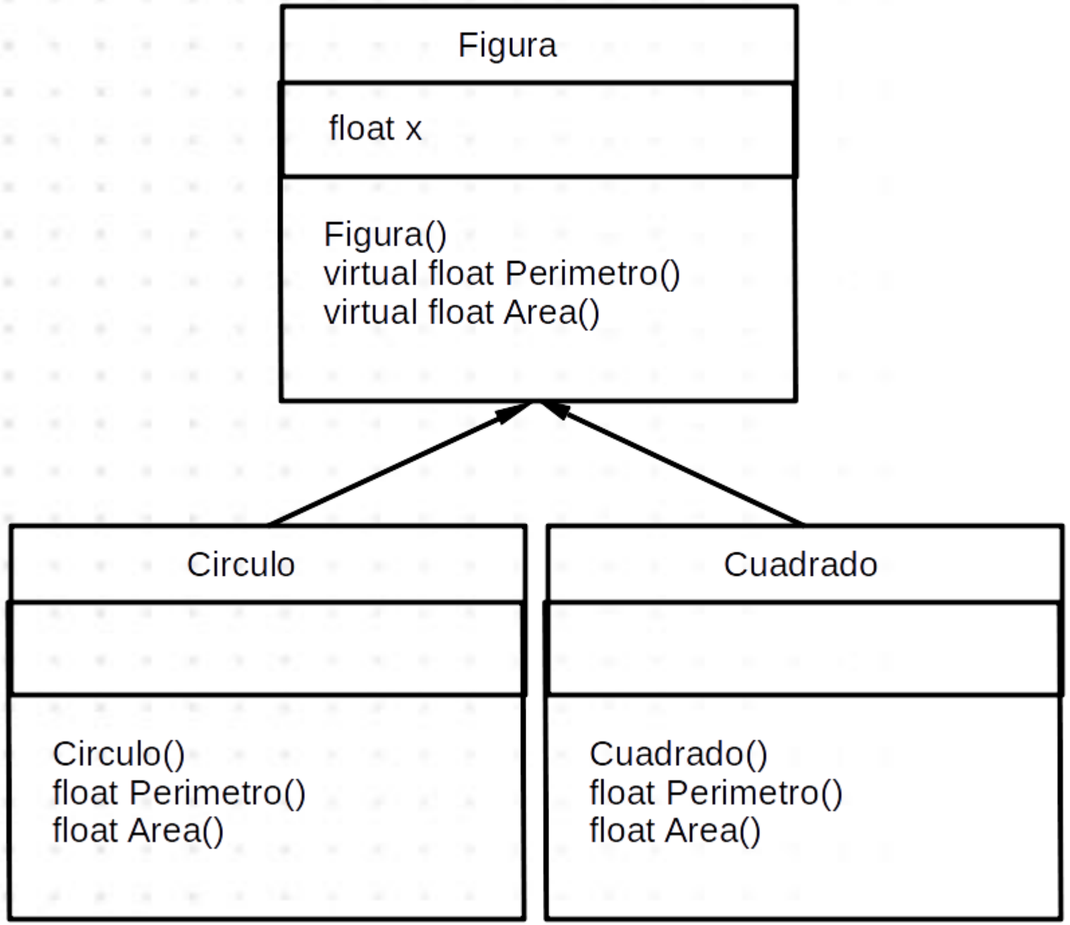
Universidad Tecnológica del Perú

Ejercicio2



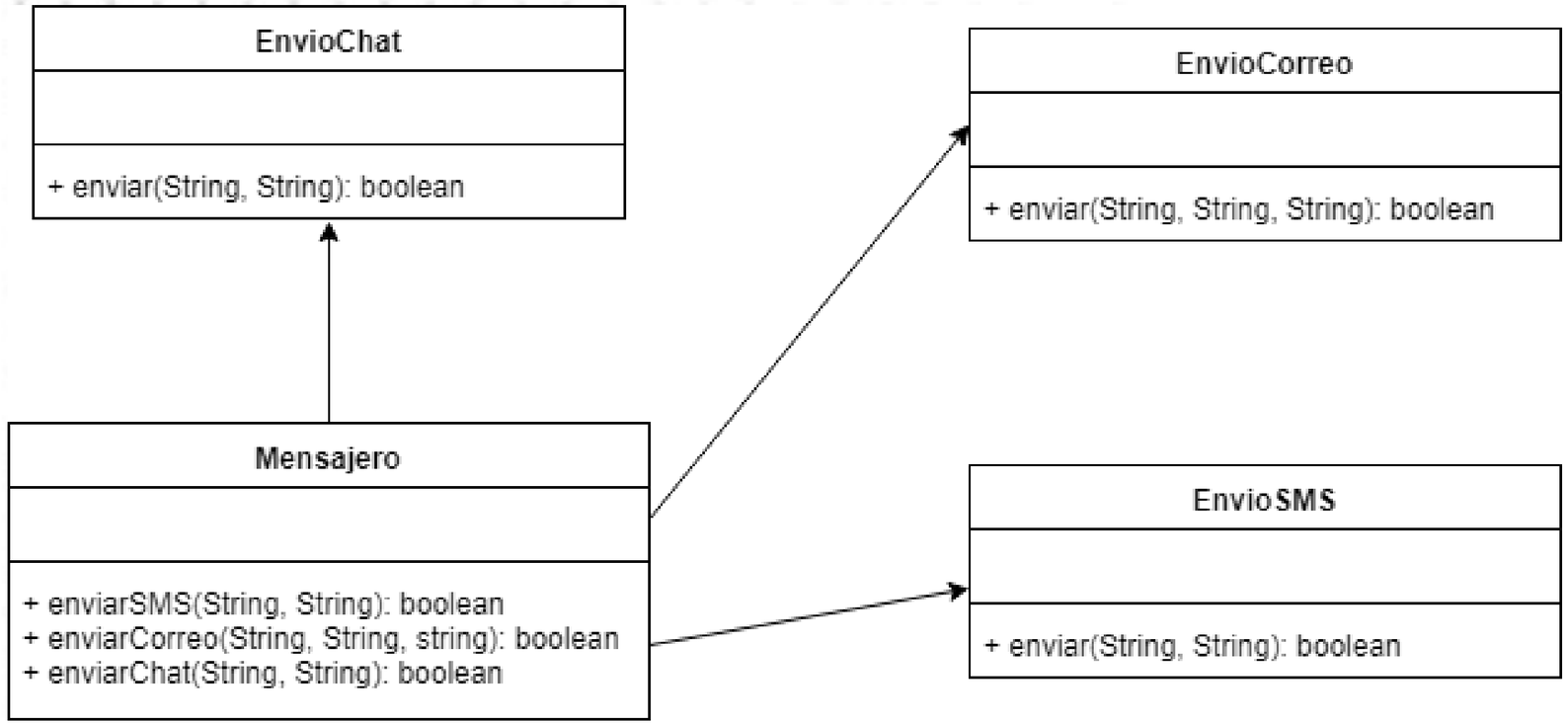
Universidad Tecnológica del Perú

Ejercicio3



Universidad Tecnológica del Perú

Ejercicio4



Universidad Tecnológica del Perú

¿Consultas o dudas?

Universidad Tecnológica del Perú

Actividad

Resolver la actividad planteada en la plataforma.

Universidad Tecnológica del Perú

Cierre

¿Qué hemos aprendido hoy?

Elaboramos nuestras conclusiones sobre el tema tratado

Universidad Tecnológica del Perú

Universidad
Tecnológica
del Perú