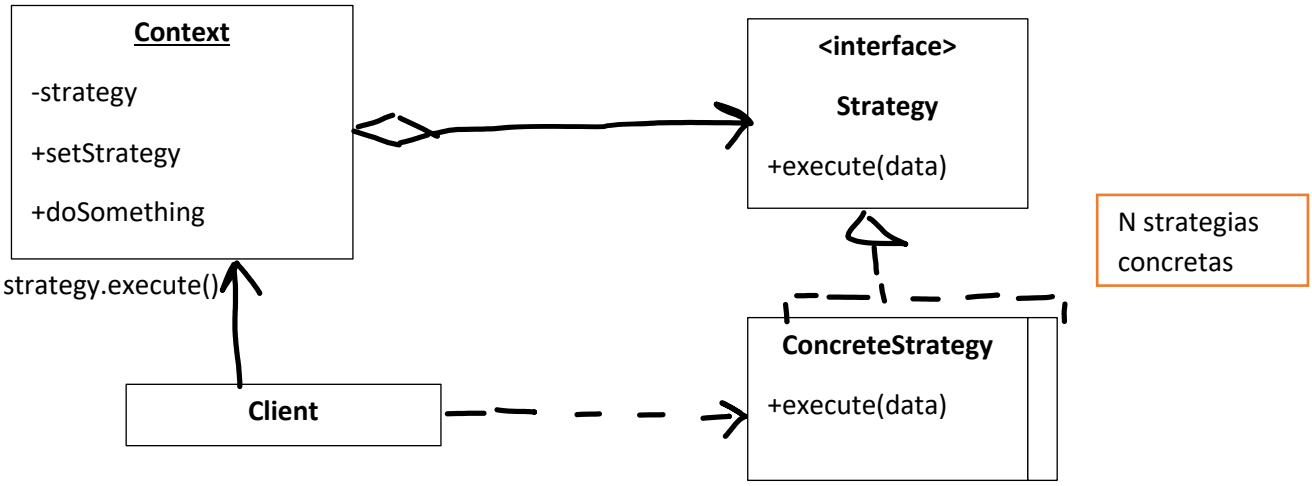
→ PREGUNTAS DE EXAMEN DE TEORÍA:

1. **Una de las consecuencias del patrón Strategy es eliminar sentencias condicionales. Ejemplo de escenario de uso, diagrama UML después de aplicar el patrón identificando cada uno de sus componentes y ejemplo de código antes y después de aplicar el patrón. Ventajas de aplicarlo frente a la herencia estática mediante polimorfismo. ¿En qué se diferencia del State?**

El patrón Strategy define una familia de algoritmos, encapsula cada uno y los hace intercambiables, de manera que el algoritmo pueda variar independientemente a los clientes que lo usan.

Este patrón se aplicaría cuando tuviésemos distintas variantes de un mismo algoritmo, de tal manera que eliminaríamos las sentencias condicionales y encapsularíamos cada variación del algoritmo (cada 'estrategia' concreta) en su respectiva clase.

El diagrama UML sería:



Componentes del patrón:

- Context(contiene una estrategia concreta que será la que se ejecute (por agregación). Podría definir métodos como getters para que la Estrategia acceda a datos del Contexto, o pasarse como referencia a esta estrategia a ejecutar.
- Strategy: interfaz común para todas las variantes del algoritmo.
- ConcreteStrategy: implementaciones específicas para cada variante del algoritmo.

Como ejemplo de aplicación del código:

Imagina que tenemos varias personas que, entre varias acciones que pueden realizar, una de ellas es hablar(). Lo más normal sería ponerlo como un método común que se realiza igual para todos, pero por ejemplo si dividimos entre PersonaEspañola y PersonaFrancesa, la española dirá "Hola" y la francesa "Bonjour". Podríamos redefinirlo en la subclase respectiva para especificar la manera en que habla distinto cada uno. Pero por ejemplo, si tenemos una persona muda, esta no hablaría, así que habría que redefinirlo una vez, y así sucesivamente. O, una persona senegalesa, que no es lo mismo que una francesa, también hablaría francés.

En vez de una herencia estática con polimorfismo, lo que se podría hacer es encapsularlo todo en una interfaz llamada 'FormaDeHablar', de tal manera que haya implementaciones como 'HablarFrancés', 'HablarEspañol' o 'NoHablar'. Entonces, en la clase base, por ejemplo, nos ahorraríamos sentencias tales como: If (persona.getNacionalidad == "francesa") { hablarFrances(); } else if (española) …..] etc, etc. Podríamos simplemente servirnos de la encapsulación y hacer: habla.hablar() en la persona, siendo habla una implementación de FormaDeHablar, que se resolverá en tiempo de ejecución.

Strategy vs. Herencia estática de polimorfismo:

Usar el patrón Strategy cuenta con una serie de ventajas, como son:

- El contexto es más fácil de entender y, más importante aún, de modificar y mantener, pues el cliente es independiente del algoritmo como tal. Por ello, si queremos cambiar la manera en que habla alguien francés, solo habría que hacerlo en la clase HablaFrances().
- Evitamos la duplicación de código.
- Evitamos la explosión de subclases a la hora de modificar y extender funcionalidades.
- Se puede cambiar dinámicamente, ya que todo depende de la interfaz.

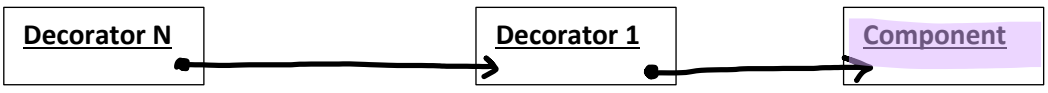
Diferencia STATE-STRATEGY:

State es como una extensión de Strategy pues ambos delegan parte del trabajo en objetos ayudantes. Mientras que en Strategy todos estos objetos son independientes entre sí, State no restringe las dependencias entre estos y puede llegar a darse acoplamiento entre ellos.

2. Diferencias entre el patrón Decorator y Strategy. Representar un diagrama de clases en tiempo de ejecución para ambos patrones.
Diferencias entre el patrón Decorator y Composite. Representar un diagrama UML identificando los participantes de cada patrón.

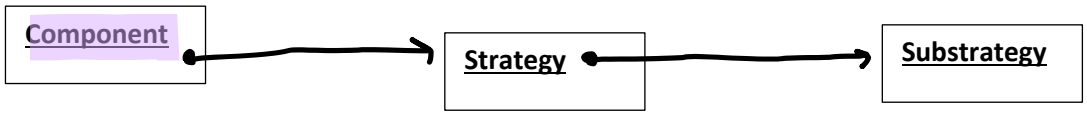
La diferencia principal (DECORATOR-STRATEGY) es que: Decorator te permite cambiar la piel de un objeto, mientras que Strategy te permite cambiar sus entrañas.

Cada decorator tendrá una referencia al respectivo componente que reviste (envuelve). A su vez, se puede aplicar más decorator cada respectivo decorator. Por tanto, el diagrama sería así:



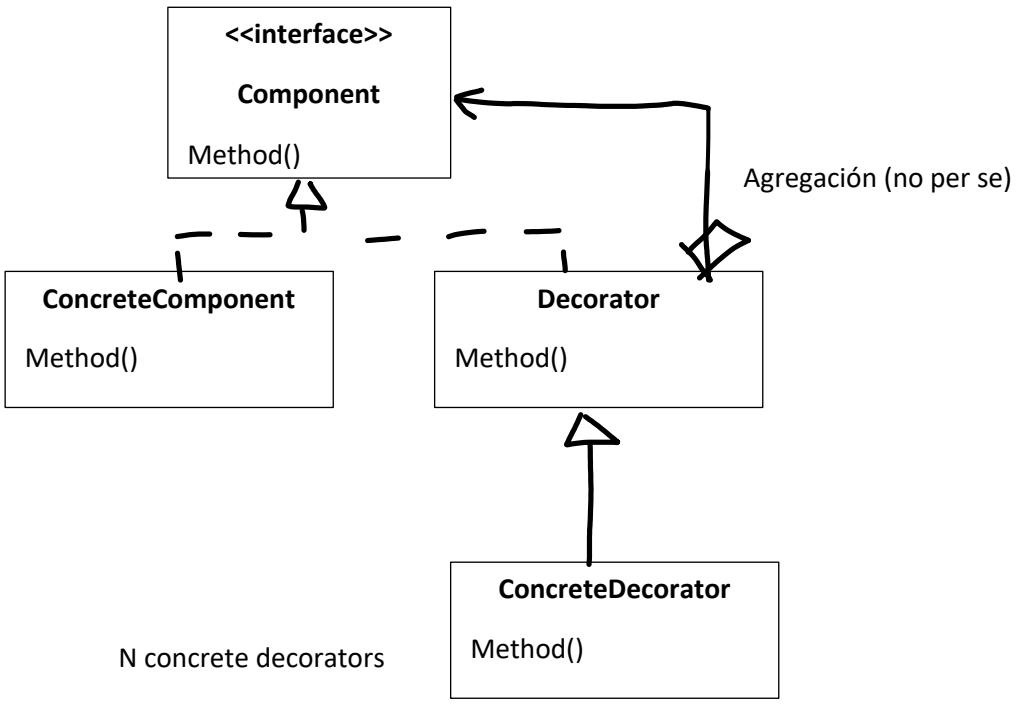
Es como si, cuando tenemos frío, nos vamos poniendo más y más capas hasta encontrar la combinación (superposición) de capas que queremos.

Como se explica en el ejercicio anterior, Strategy cambia las entrañas. Es decir: encapsula el comportamiento que varía en su respectiva clase para independizarlo del cliente que lo usa. Los detalles de implementación son ajenos a la clase que interactúa con la interfaz que lo encapsula a todo. Por tanto, el diagrama sería:

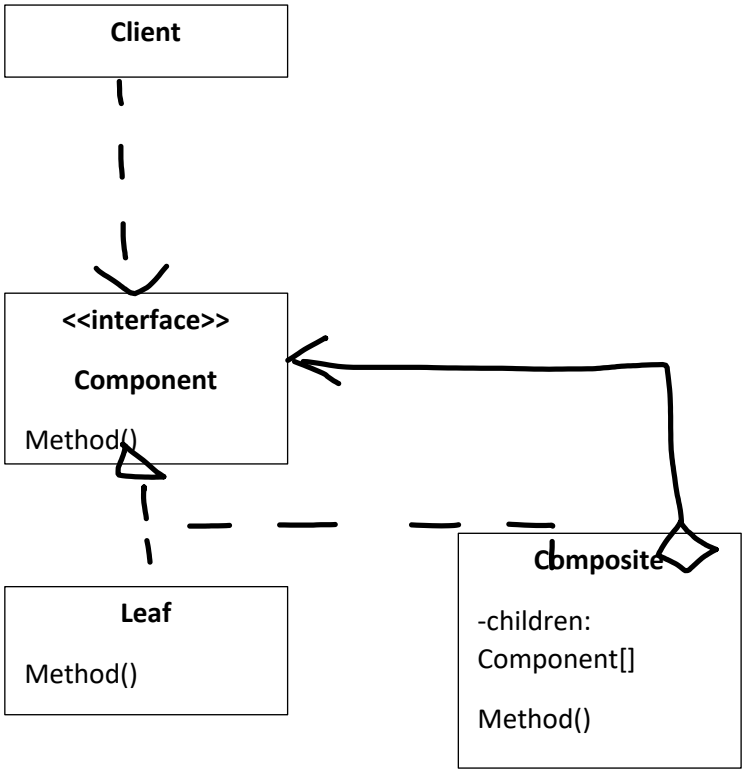


(es decir, el flujo va al revés).

La principal diferencia (DECORATOR-COMPOSITE) es que un Decorator es un compuesto pero con **un solo componente**. Además, Decorator añade responsabilidades adicionales sobre el objeto envuelto, no está pensado para agregar objetos en un compuesto (como un Composite, que tiene varios componentes) El diagrama UML del Decorator sería:



Y el de Composite



3. **Tenemos un reproductor para cada tipo de elemento multimedia: canciones y videos. Aplicar un patrón para eliminar las sentencias condicionales. Identificar el patrón aplicado, estructura y componentes, cómo quedaría el código al eliminar las sentencias condicionales.**

```
public class ListaDeReproduccion { ...
      public void reproducir() {
      Iterator<Elemento> iterador = elementos.iterator();
      while (iterator.hasNext()) {
            Elemento elemento = iterator.next();
            Reproductor reproductor;
            if (elemento instanceof Video)
                  reproductor = new ReproductorDeVideo();
            else if (elemento instanceof Cancion)
                  reproductor = new ReproductorDeAudio();
            reproductor.reproducir(elemento);
      }
}
```

Aquí vemos que la sentencia condicional se realiza para la creación de un reproductor para cada tipo de elemento. Si el elemento es de tipo vídeo, se creará un Reproductor de Vídeo. Por otro lado, si es audio, se creará un reproductor de audio.

Para eliminar las sentencias condicionales, deberíamos encapsular el código relacionado con la creación de un reproductor determinado.

Para ello podríamos aplicar el **FactoryMethod**: desacoplar la creación de objetos de la implementación que usa dichos objetos. Los elementos que participan son:

- Un 'Product': la interfaz común de los objetos creados.
- Varios 'ConcreteProduct': las implementaciones de cada producto.
- 'Creator': la factoría que, con un método abstracto, crea cada producto de la Interfaz.
- Varios 'ConcreteCreator': redefinición de la implementación de fábrica para devolver un ConcreteProduct.

El Product sería la interfaz Reproductor, con un método reproducir(e: Elemento). Los ConcreteProduct serían ReproductorDeVideo y ReproductorDeAudio, con las implementaciones del método reproducir(e:Elemento). El 'Creator' tendría un método abstracto de creación que devolvería un 'Reproductor' y los 'ConcreteCreator' devolverían cada uno el Reproductor específico de cada implementación. Por ejemplo:

Teniendo en cuenta que ReproductorDeVideo y ReproductorDeAudio son implementaciones de la interfaz 'Reproductor':

```
public class ListaDeReproduccion { ...
      public void reproducir() {

            Iterator<Elemento> iterador = elementos.iterator();
            while (iterator.hasNext()) {
                  Elemento elemento = iterator.next();
                  Reproductor reproductor =
                  ReproductorFactory.crearReproductor(elemento);
                  reproductor.reproducir(elemento);
```

```
                }
        }
```

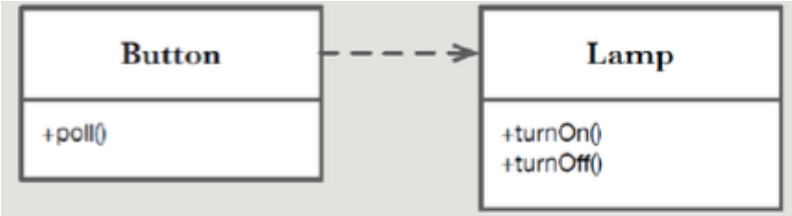
En una variante del patrón, tenemeos una **factoría parametrizada**, que construye Reproductores de dos tipos. Realmente no es un patrón como tal ya que lo único que estamos haciendo es pasar estas sentencias condicionales a otra clase.

```
public abstract class ReproductorFactory { ...
            public abstract static Reproductor
                  crearReproductor(Elemento e) {

                        if (elemento instanceOf Video) {
                              return new ReproductorDeVideo();
                        } else {
                              Return new ReproductorDeAudio();
                        }
                  }
   }
```

O bien podríamos hacer → hacer que cada elemento cree su reproductor:

```
      public class ListaDeReproduccion { ...
            public void reproducir() {

                  Iterator<Elemento> iterador = elementos.iterator();
                  while (iterator.hasNext()) {
                        Elemento elemento = iterator.next();
                        Reproductor reproductor =
                        elemento.crearReproductor();
                        reproductor.reproducir(elemento);
                  }
         }

      public interface Elemento { ...
            public void reproducir();
            public Reproductor crearReproductor();
         }

      public class Video implements Elemento { ...
            public void reproducir() {  … }
            public Reproductor crearReproductor(){
                  return new ReproductorDeVideo(); }
            }
      }

      public class Audio implements Elemento { ...
            public void reproducir() {  … }
            public Reproductor crearReproductor(){
                  return new ReproductorDeAudio(); }
            }
      }
```

4. **Define el <u>principio de inversión de dependencias</u>. Aplícalo para mejorar el siguiente diseño en el que un botón permite encender o apagar una lámpara (representa cómo quedaría el diagrama de clases tras aplicarlo). ¿Qué se ha logrado?**



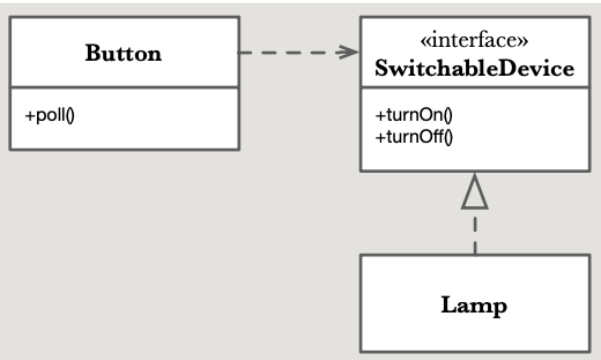

```
public class Button
{
    private Lamp lamp;
    public void poll()
    {
        if (/* some condition */)
            lamp.turnOn();
    }
}
```

El **principio de inversión de dependencias (DIP**) es uno de los principios SOLID, que dice que los módulos de alto nvel no deben depender de los de bajo nivel: ambos deben depender de abstracciones.

Un ejemplo de su uso es el siguiente:

Aquí, vemos cómo el estado de la lámpara (un sistema mayor) depende de algo más pequeño (un botón, con una implementación específica). La conexión entre módulo de alto nivel y módulo de bajo nivel debería estar regulado por una interfaz, ya que un botón no sirve solo para una lámpara. Existen muchos más tipos de elementos que se pueden apagar/encender con un botón.

Por tanto:



Gracias a esto, el botón puede apagar y encender cualquier elemento que implemente esta interfaz.

5. **Completar esta tabla:**

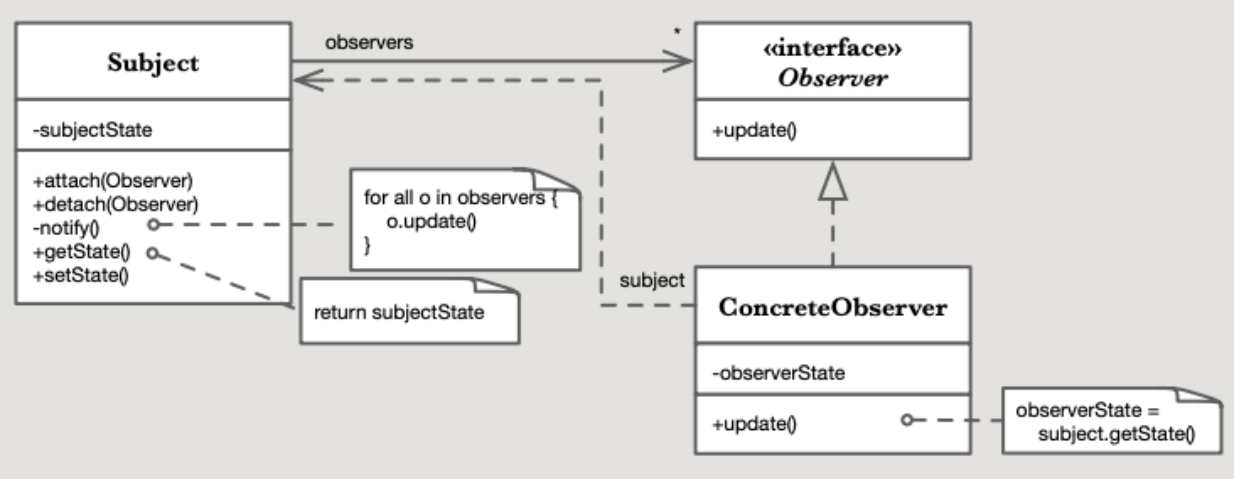
	INTERFAZ	**CLASE ABSTRACTA**
Situación en que se originan	Extracción de responsabilidades	Facilitar nuevas implementaciones, reutilizar código y evitar su duplicación
Creador	Analista/Diseñador	Programador
Raíz	Sí	No
Importancia	Son fundamentales: son el enlace dinámico, lo que nos permite eliminar lógica condicional	Comodidad (es un mero copy-paste)
Operaciones	Añaden funcionalidad a sus ancestros	Implementan la funcionalidad de sus ancestros, y añaden operaciones de implementaciones parciales
Importancia de los cambios	Alta: obligan a los programadores a cumplir un cierto contrato	Baja: la interfaz hace de cortafuegos

6. **<u>Patrón Observer</u>: cuál es su intención, cuándo es aplicable, qué se consigue al aplicarlo. Representa la estructura del patrón mediante un diagrama UML y explica el papel de los distintos participantes y los métodos relevantes del patrón. Igualmente, dibuja un diagrama de secuencia donde se vea un escenario de uso típico del patrón en tiempo de ejecución, representando todas las colaboraciones entre los objetos participantes.**

El **patrón Observer es un patrón de diseño de comportamiento** que permite definir un mecanismo de suscripción para notificar a varios objetos osbre cualquier evento que le suceda al objeto que están observando. Es decir: una dependencia one-to-many de tal manera que, cuando el uno cambie su estado, todos los many se actualizan automáticamente.

Este patrón se aplican cuando los cambios de estado en un objeto necesitan cambiar otros objetos y este grupo sea desconocido o cambie dinámicamente. Por ejemplo, tenemos el caso de una interfaz gráfica con sus numerosos 'listeners' de eventos.

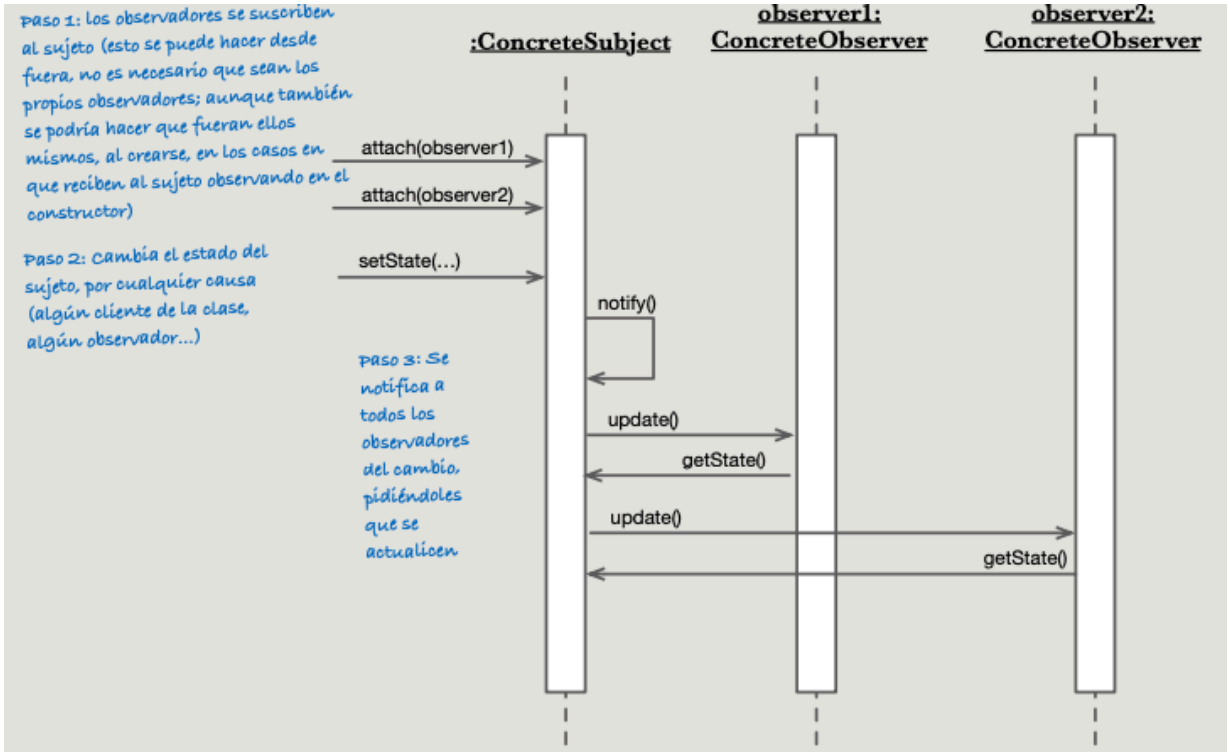
El diagrama UML del patrón Observer sería:



Los componentes son:

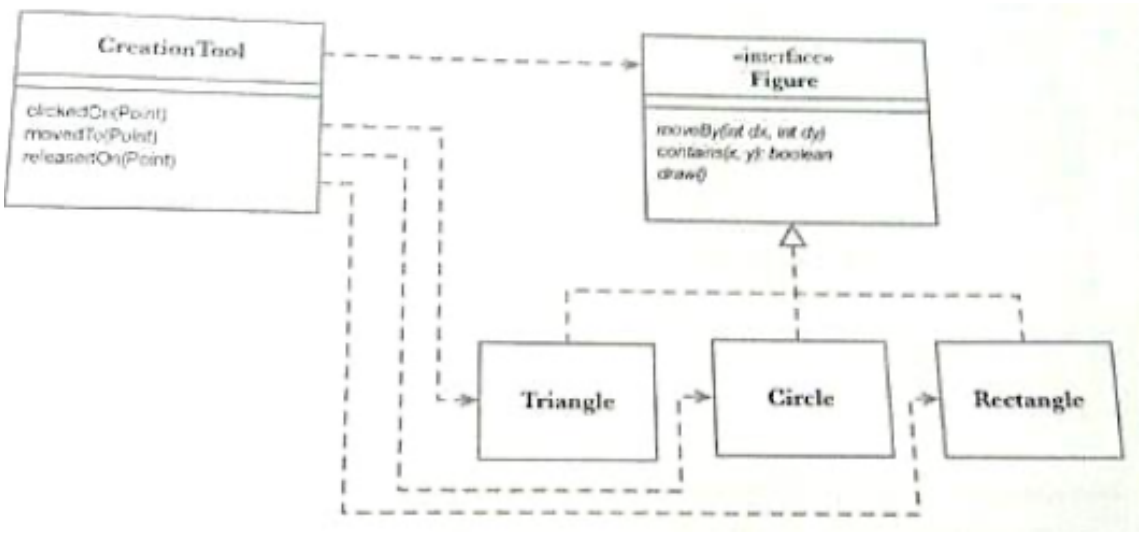
- Subject: interfaz general de clase observada para se suscriban o borren los observadores.
- Observer: interfaz que define el método de actualización de todos los observadores concretos.
- ConcreteSubject: es el objeto observado en un determinado estado. Envía una notificación a los observadores siempre que se modifica su estado.
- ConcreteObserver: implementaciones de la actualización de los observadores. Tienen una referencia al Subject observado y son actualizados siempre que el sujeto les manda la actualización.

El diagrama de secuencia sería:



(caso de que el subject tenga 2 observadores). Se añaden los 2 observadores y, cada vez que se cambia el estado, se manda la notificación, en la que se actualiza cada observador.

7. **A continuación, se representa el diseño actual de la parte de editor gráfico encargada de crear los distintos tipos de figuras, junto con un esbozo del código de la clase CreationTool. ¿Qué hay de malo en el código? Aplica un patrón que permita eliminar la lógica condicional a la hora de crear unas figuras u otras empleando polimorfismo. Representa un diagrama de clases con la solución y escribe cómo quedaría el código con el nuevo diseño (tanto la clase original como al menos el correspondiente a la creación de una figura determinada, así como la creación de los tres objetos que representan las distintas herramientas de creación). ¿De qué patrón se trata? ¿Qué se ha conseguido al aplicarlo?**




```
public class CreationTool {
     public enum FigureType {RECTANGLE, CIRCLE, TRIANGLE}
     private FigureType type;
     private Editor editor;
     private Point initialPoint;

     public void releasedOn(Point endPoint) {
          Figure newFigure;
          if(type == FigureType.RECTANGLE)
               newFigure = new Rectangle
               (initialPoint,endPoint);
          else if(type == FigureType.CIRCLE) {
               newFigure = new Circle
               (initialPoint,endPoint);
          ... TRIANGLE ...
          editor.getDrawing().addFigure(newFigure);
          editor.toolDone();
     }
}
```

Mientras que la creación de los tres objetos representando las distintas herramientas de creación, en esta primera versión, sería algo así como:

```
CreationTool rectangleTool = new CreationTool(CreationTool.
     FigureType.RECTANGLE);
CreationTool circleTool = new CreationTool(CreationTool.
     FigureType.CIRCLE);
CreationTool triangleTool = new CreationTool(CreationTool.
     FigureType.TRIANGLE);
```

Haz lo mismo con el patrón Prototype.

Diría que lo malo de este código es que tenemos acoplamiento entre la figura que se crea y el editor de dibujo. Esto se ve plasmado en que estamos haciendo sentencias condicionales en el

método principal de la herramienta de creación. Si quisiésemos extender la funcionalidad y añadir más figuras, nos veríamos obligados a realizarlo en ese método. La mejor opción sería encapsular toda esa lógica de creación en una clase aparte con esa responsabilidad propia (la de crear figuras en base a un tipo).

Entonces, lo que podríamos hacer es con el **FACTORY METHOD**:

```
CreationTool rectangleTool = new CreationTool(new RectangleFactory());
CreationTool circleTool = new CreationTool(new CircleFactory());
CreationTool triangleTool = new CreationTool(new TriangleFactory());

public abstract class FigureFactory {
      public abstract Figure createFigure(initialPoint,endpPoint);
}

Public class RectangleFactory extends FigureFactory {
      Public Figure createFigure(initialPoint,endPoint) {
            return new Rectangle(inirialPoint,endPoint);
      }
}

Public class CreationTool {
      Private Factory factory;
      Private Editor editor;
      Private Point initialPoint;

      Public CreationTool(FigureFactory f) { this.factory = f; ]

      Public void releasedOn(Point endPoint) {
            Figure f = factory.createFigure(initialPoint,endPoint);
            Editor.getDrawing().addFigure(newFigure);
            Editor.toolDone();
      }
}
```

Con el **PATRÓN PROTOTYPE**:

```
Public interface Figure {
      Figure clone();
}

Public class Rectangle implements Figure {
      Private int width;
      Private int height;
      Public Rectangle(Rectangle r) {
            This.width = r.width;
            This.height = r.height;
      }
      Public Figure clone() {
            Return new Rectangle(this);
      ]
]

Public class CreationTool {

      Private Figure figureType;
      Private Editor editor;
      Private Point initialPoint;
      Public CreationTool(Figure f) { this.figureType = f; ]

      Public void releasedOn(Point endPoint) {
            Figure newFigure = figureType.clone();
            newFigure.setDimensions(initialPoint,endPoint);
            (...)
      ]
]
```

8. Definir el patrón Decorator. Diferencias entre Decorator – Strategy – State.

El patrón Decorator es un patrón estructural que nos permite añadir funionalidad extra a un objeto envolviendo a este en un envoltorio especial que contiene nuevos comportamientos. Dentro de su estructura se encuentra una interfaz Componente con una determinada implementación. Parte de esta interfaz componente es la interfaz Decorador, que contiene una referencia al componente y así le añade funcionalidad extra a través de las implementaciones concretas de decorador.

Aplicamos este patrón cuando necesitamos asignar comportamiento extra dinámicamente sin romper y explotar la estructura del código de estos objetos. Es decir, nos permite ir añadiendo capas sin modificar lo que ya existe, lo cual nos permite extender el comportamiento sin utilizar herencia.

- Diferencia con strategy: el patrón Decorator nos ayuda a revestir y envolver un objeto añadiéndole funcionalidad extra dinámicamente. Sin embargo, Strategy nos permite modificar el comportamiento desde dentro, delegando el trabajo en subclases concretas que implementan cada una de las variantes de un algoritmo.
- Decorator y State son prácticamente diferentes: State sería prácticamente lo mismo que Strategy en cuanto a diferencias, salvo que State no restringe el acoplamiento y dependencia entre las subimplementaciones.

9. Práctica 'Ball Game': proponer soluciones con 2 patrones distintos.

Ball Game es un juego que se podía sacar para PS5, Xbox o PC. Existirían 2 maneras de plantearlo:

- Con un Template Method para cada juego, se hacen clases hijas con la implementación abstracta que difiere del algoritmo esqueleto.
- Con el patrón Adapter creamos un adaptador (como un enchufe) entre el cliente del juego y la implementación concreta de cada plataforma.

10. Define el principio de abierto-cerrado y di qué patrón lo ejemplifica.

El principio de abierto cerrado dice que las clases deberían estar abiertas para la extensión pero cerradas para la modificación: es decir, podemos aprovechar lo que ya existe y extenderlo añadiendo funcionalidad, pero no modificar su base.

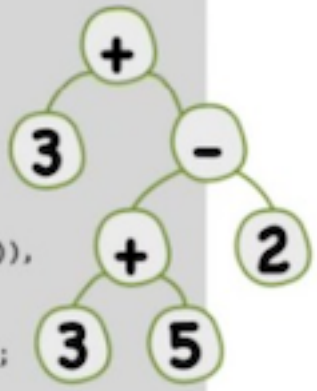
Por ejemplo: digamos que estamos en una empresa que trabaja para la sección de Google Maps y hemos de añadir nueva funcionalidad a una aplicación existente que sigue en desarrollo por otro equipo o se sigue usando. No debemos modificar lo ya hecho, debemos reutilizarlo y extenderlo. O, por ejemplo, las librerías de software.

Un patrón que lo ejemplifica es el patrón State: la ejecución depende del estado del objeto en ese momento, y este comportamiento concreto según el estado está encapsulado en una clase concreta, independiente del resto del código, e implementando una interfaz concreta. El objeto que interactúa con esta interfaz no conoce los detalles, y es muy fácil modificar o extender el comportamiento concreto del objeto en ese momento.

4. Aplicar el patrón Visitor a esta clase.

```
public interface Expression
{
}

public class Number implements Expression
{
   public int value;

   public Number(int value) { ... }
}

public class Sum implements Expression
{
   public Expression left, right;

   public Sum(Expression left, Expression right) { ... }
}

public class Subtraction implements Expression
{
   public Expression left, right;

   public Subtraction(Expression left, Expression right) { ... }
}
```

```
public class Main
{
   public static void main(String[] args)
   {
        // 3 + ((3 + 5) - 2)
        Expression expression = new Sum(
             new Number(3),
             new Subtraction(
                  new Sum(new Number(3), new Number(5)),
                  new Number(2)));

        Visitor interpreter = new InterpreterVisitor();

        int result = _____;

        System.out.println(result);
   }
}
```



```
Public interface Expression {
      public void accept(Visitor v);
]
Public class Number implements Expression {
      Public void accept(Visitor v) { v.visitNumber(this); ]
]
Public class Sum implements Expression {
      Private Expression left,right;
      Public void accept(Visitor v) { v.visitSum(this); ]
]
Public class Substraction implements Expression {
      Private Expression left,right;
      Public void accept(Visitor v) { v.visitSub(this); ]
]
Public interface Visitor { visitNumber, visitSum, visitSub… ]
Public class InterpreterVisitor implements Visitor {
      Public void visitNumber(Number n) {
            Return n.value;
      ]
      Public void visitSum(Sum s) {
            Return s.left.accept(this) + s.right.accept(this);
      ]
      Public void visitSub(Substraction s) {
            Return s.left.accept(this) - s.right.accept(this);
      ]
]

Public void main(String[] args) {
      Visitor v = new InterpreterVisitor();
      Int result = expression.accept(v);
]
```

→ EJERCICIOS DE SEMINARIO:

HERENCIA

1. **¿Debería Stack derivar de Vector?**

   Si derivase de Vector, estaría obligada a cumplir el contrato que especifica la funcionalidad y responsabilidad propia de un vector (ej. ArrayList). Puede no compartir la funcionalidad.
   Una jerarquía a partir de una interfaz no debe de surgir como una clasificación, debe surgir como una necesidad de implementar una determinada funcionalidad.

2. **Un avión puede estar en hangar, pista, a punto de despejar o aterrizar, en el aire… En cada una de estos momentos queremos realizar una u otra operación. ¿Cómo?**

   Un diseño del estilo <interfaz> Avión con sus implementaciones AvionEnPista, AvionEnHangar, AvionVolando… es decir, implementaciones específicas para cada caso: no es la mejor opción. Lo que podríamos hacer es utilizar el patrón State para especificar la funcionalidad concreta en cada estado del avión. Todo se encapsula en una interfaz que emplea un método común, cuya implementación dependerá dinámicamente del estado del avión.

3. **¿Hacemos una jerarquía para diferenciar entre sensores de temperatura y de humedad?**

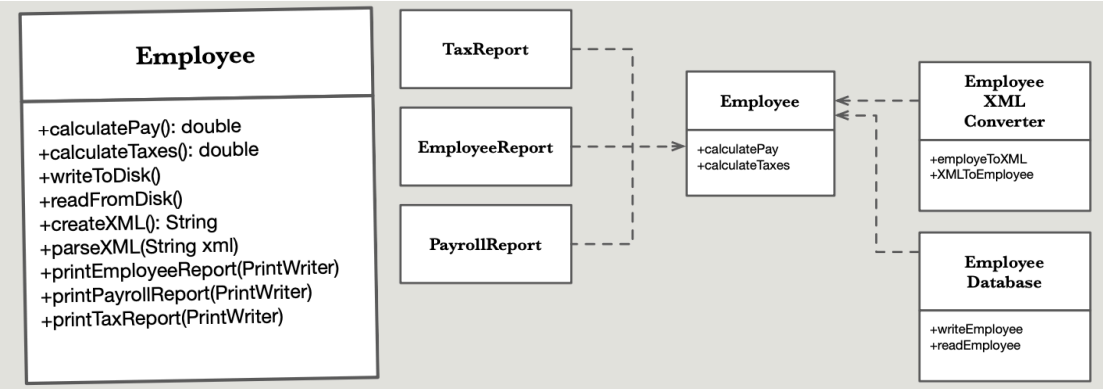
   Pensemos en: ¿se pasará un sensor de temperatura indistintamente de un sensor de humedad? Es decir, ¿se pasará sin que importe cuál es? → No.
   Realmente no es una herencia: son dos responsabilidades distintas, no una funcionalidad concreta cuyo funcionamiento varía.
   Por tanto, una herencia de Sensor no sería correcta.

4. **¿Cuadrado hereda de Rectángulo?**

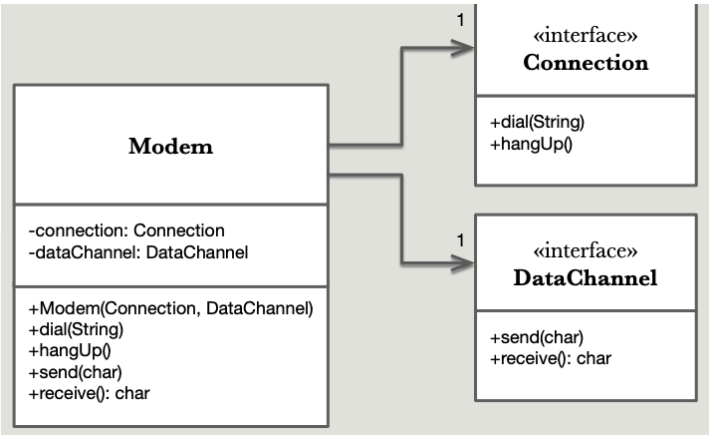
   Podríamos hacer fácilmente (y no elegantemente) que Cuadrado herede de la superclase rectángulo sus métodos setHeight y setWidth, poniendo siempre el mismo valor para ambos. Así, cuando su altura cambia, su ancho también.
   Sin embargo, habrá momentos en que no se cumpla el principio de sustitución de Liskov: no siempre se puede sustituir indistintamente cuadrado por rectángulo (es decir, si queremos asertar que el área es 20 para un rectángulo de 4x5, no lo será para un cuadrado).

PRINCIPIOS SOLID

5. Principio de Responsabilidad Única:




```
public interface Modem
{
    void dial(String number);
    void hangUp();
    void send(char c);
    char receive();
}
```



6. Principio de Abierto-Cerrado:

Imagina que tenemos una pizarra donde dibujar figuras. Cada vez que añadamos una nueva, tendremos que ejecutar todas las sentencias condicionales en todos los momentos donde se usen dichas clases, por lo que el impacto es enorme.

```
class Pizarra
{
  private Figura[] figuras = new Figura[30];
  private int contador = 0;

  public void añadirFigura(Figura figura)
  {
    figuras[contador++] = figura;
  }

  public void dibujar()
  {
    for (int i = 0; i < contador; i++) {
      figuras[i].dibujar();
    }
  }
}
```

7. **LSP (Principio de sustitución de Liskov):**

Reconoceremos este principio cada vez que damos una implementación vacía o es ilegal invocar un método en un tipo derivado (ej. obligando a todos los métodos derivados a lanzar/capturar una excepción debido a la implementació ilegal de uno de ellos).

Ej: imagina que queremos añadir un empleado voluntario.

Si determinamos que su calcPay() == 0, estamos diciendo que tiene sentido calcular la paga de este voluntario en sí, cosa que no es verdad.

En caso de que todo esto se cumpla, es señal para ver que no hemos de aplicar herencia en este caso.

LOGGER (PATRÓN SINGLETON)

8. **¿Cómo garantizar que una clase tenga una única instancia?**

Hemos de evitar que absolutamente nadie pueda instanciar a new() desde la clase → es decir, que tenga el constructor privado. Así creamos un único logger de consola:

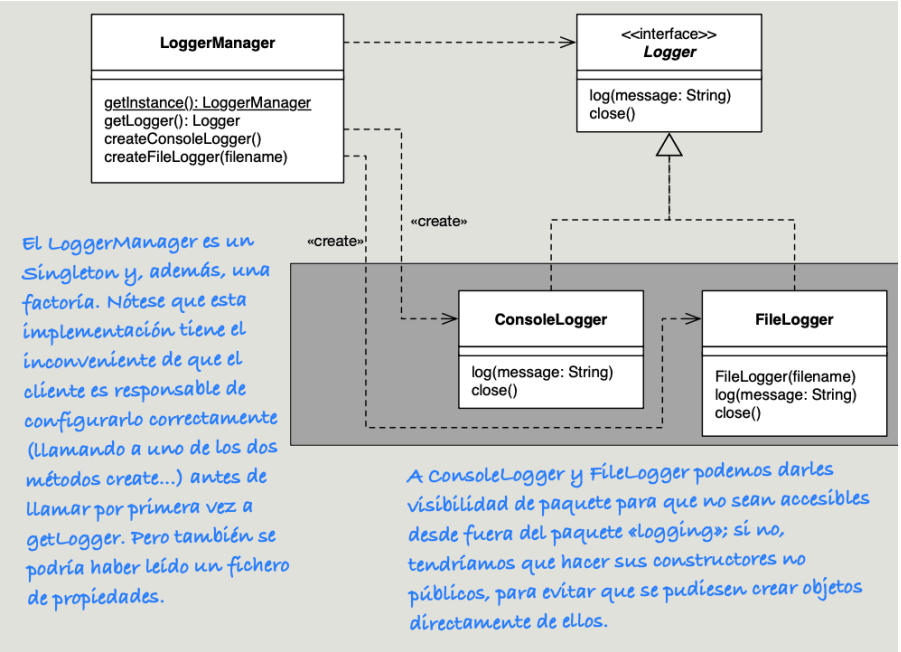
El **patrón Singleton** es un patrón de creación que se asegura de que una clase tiene una única instancia, a la que nos da acceso global.

```
Public class Logger {

      Private static final Logger l = new Logger();
      Private Logger() { }

      Public static Logger getLogger() {
            Return l;
      ]

      Public void log(String m) { System.out.println(m); }
]
```

9. **Existen 2 tipos de Logger: consola y fichero. Una vez decidido el tipo, ya no se podrá cambiar. Ha de haber una sola instancia única.**

Existe la interfaz 'Logger' que proporciona la funcionalidad básica de loguear un mensaje. Tiene 2 implementaciones concretas: por consola o por fichero.

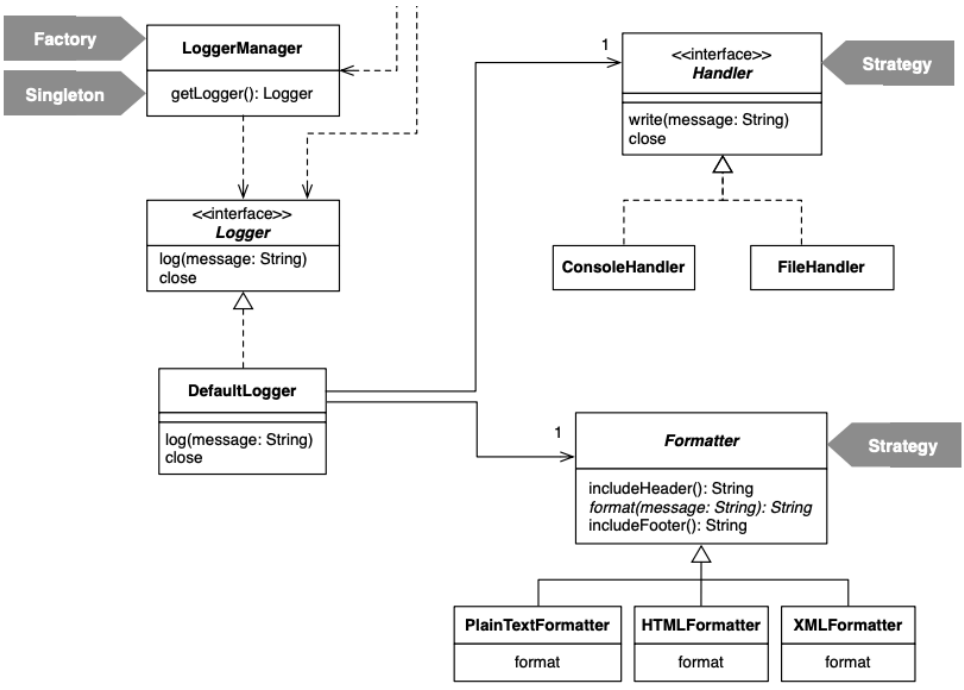
Ha de haber una sola instancia de un tipo concreto de logger. Esto será controlado por el LoggerManager, que será un Singleton + Factoría que crea al logger.



Para crear el tipo de logger podríamos hacer un método de creación para cada logger o bien hacer un solo método parametrizado por el tipo de logger a crear.

Entonces: el logger es un singleton, y luego el logger manager es un singleton también, así que nos aseguramos de que solo haya una instancia de un tipo concreto.

Luego, si quisiésemos añadir funcionalidad que puede ir cambiando, podríamos aplicar el patrón Strategy:

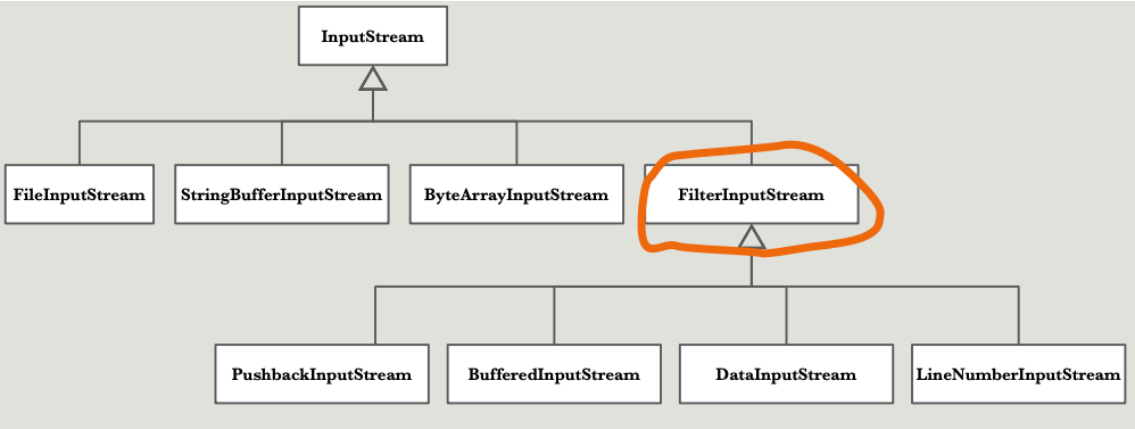


EJERCICIOS DE PATRONES

10. **Queremos hacer un reproductor multimedia tipo iTunes que puede contener: canciones, listas de reproducción (con canciones) o carpetas (con carpetas y listas de reproducción):**

→ Patrón **COMPOSITE**

11. **Filtros en Java I/O**



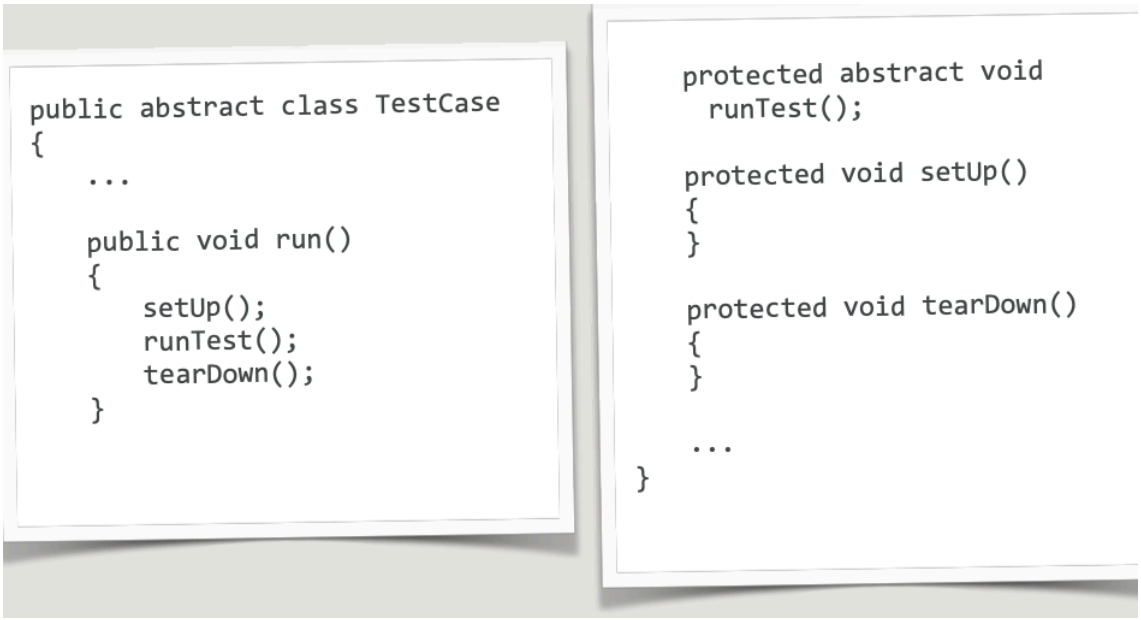
Todas las clases que heredan de FilterInputStream son implementaciones de decoradores concretos. Si quisiésemos implementar un decorador convierte todo el I/O a lowercase:

```
Public class LowerCaseInputStram extends FilterInputStream{
      Public LowerCaseInputStream(InputStream in) { super(in); ]
      Public int read() throws IOException {
            Int c = super.read();
            Return (c == -1 ? c: Character.toLowerCase((char)c));
      ]
```

] → Patrón **DECORATOR**

12. **Un framework de pruebas implementa en su clase TestCase un método run(): luego los programadores han de implementar dicho método en sus propias pruebas unitarias. Quieren que siempre se ejecute un método de inciialización primero y luego una liberación de recursos antes del run.**

→ Patrón **TEMPLATE METHOD**.



```
public abstract class TestCase
{
    ...

    public void run()
    {
        setUp();
        runTest();
        tearDown();
    }
```

```
    protected abstract void
      runTest();

    protected void setUp()
    {
    }

    protected void tearDown()
    {
    }

    ...
}
```