Patrones de diseño

Dra. Amparo López Gaona
Fac. Ciencias, UNAM

Febrero-Marzo 2013

Introducción

- El desarrollo del software es una tarea complicada que depende en gran medida de la **experiencia** de las personas involucradas.
  Mecanismos de reutilización:
  - Herencia.
  - Agregación.
  - Genericidad.
    Estos tres son a nivel código
  - Patrones de diseño. (A nivel estructura)
  - Un patrón de diseño describe la solución general, probada, a un problema común que ocurre en el desarrollo de software.
  - La descripción de esta solución está formada por un conjunto pequeño de clases y sus interacciones.

En la vida diaria: patrones de conducta:

Patrones de diseño

Un patrón de diseño es:

- Una sollución estándar para un problema común de programación
- una técnica para flexibilizar el código haciéndolo satisfacer ciertos criterios
- una manera práctica de describir ciertos aspectos de la organización de un programa

Clasificación

Se clasifican para facilitar la búsqueda del más adecuado, de acuerdo a dos criterios:

- Propósito. Reflejan lo que realiza el patrón.
  - Creación.
  - Estructural.
  - Comportamiento.
- Alcance. Indica si el patrón se aplica a clases u objetos.

... Clasificación

	Creación	Estructural	Comportamiento
Clase	Método de fabricación	Adaptador	Interpréte Plantilla
Objeto	Fábrica Constructor Prototipo Singleton	Adaptador Puente Composición Decorador Fachada	Cadena de responsabilidad Comando Iterador Intermediario Observador Estrategia

Descripción

Para la descripción de los patrones se incluye por lo menos:

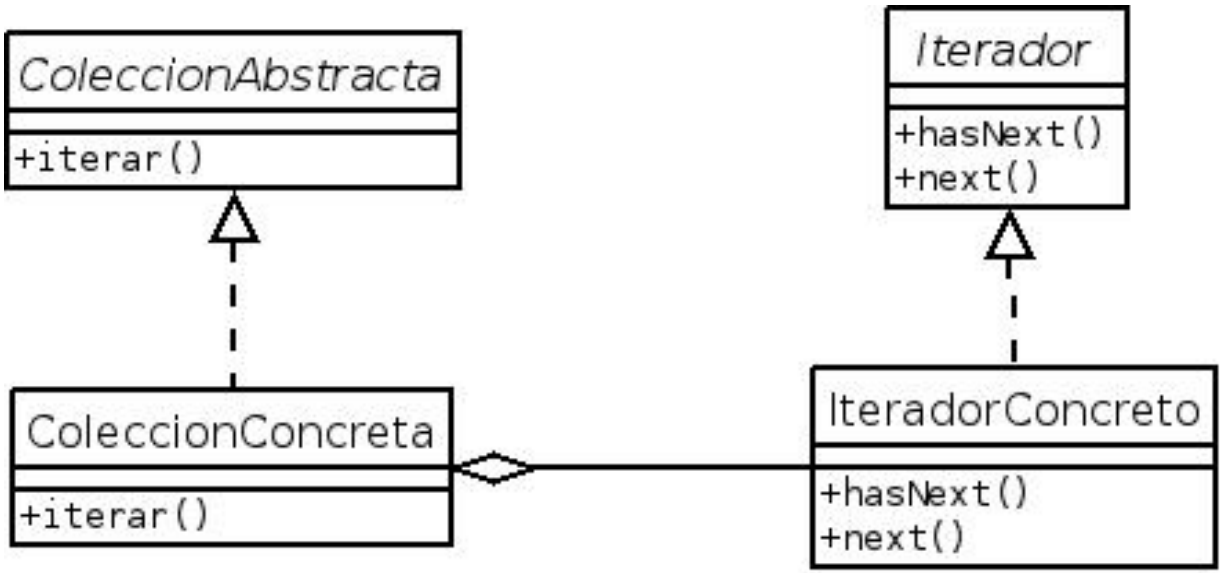
- El nombre del patrón. Describe el problema de diseño, su solución, y consecuencias en una o dos palabras.
- Una breve descripción del problema que ataca. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño rígido Se incluye una lista de condiciones.
- La descripción de la solución. Describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular. Un patrón es una plantilla
- Las consecuencias de usar el patrón, incluyendo el resultado y compromisos.

Ejemplo de patrón de diseño

Recorrer una colección de datos.

- Nombre: Iterador
- Categoría: Patrón de diseño de comportamiento.
- Objetivo: Proporcionar una forma de accesar secuencialmente los elementos de una colección.
- Aplicabilidad: El iterador debería usarse para:
  - Accesar el contenido de una colección sin exponer la representación inerna.
  - Soportar recorridos múltiples de las colecciones. Por ejemplo, recorridos anidados.
  - Proporcionar una interfaz uniforme para recorrecorer diferentes colecciones y con ello soportar iteración polimorfica.

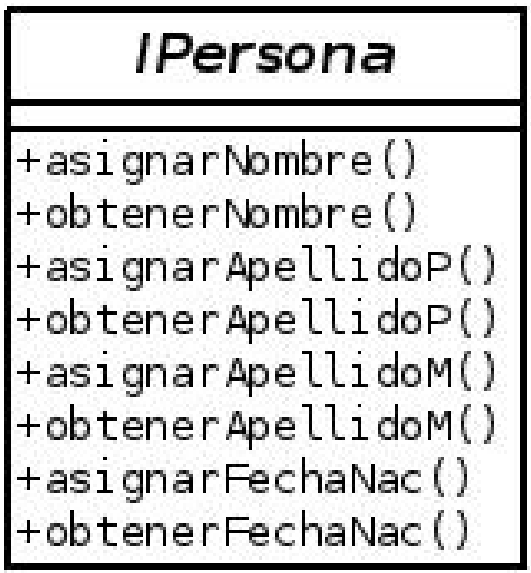
... Iterador

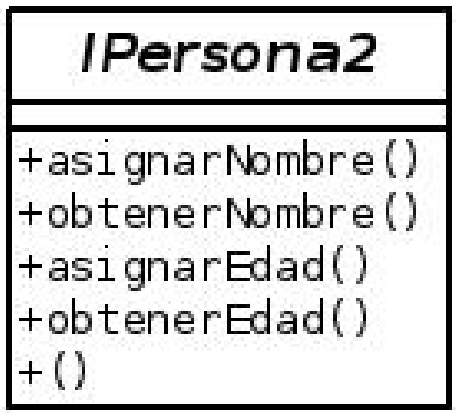


- Los participantes en este patrón son:
  - `Iterador`, define una interfaz para acceder y recorrer los elementos.
  - `IteradorConcreto`, implementa la interfaz y conserva la posición actual en el recorrido de la colección.
  - `ColeccionAbstracta`, define una interfaz para crear un iterador concreto.
  - `ColeccionConcreta` implementa y devuelve una instancia de un iterador concreto apropiado.

Patrón adaptador

Suponer que se tiene una aplicación que tiene una clase `Persona` que implementa la interfaz `IPersona`
Pasado el tiempo se requiere que la aplicación también trabaje con objetos de otra clase que implementa la interfaz `IPersona2`, que aunque es similar a `IPersona2`.

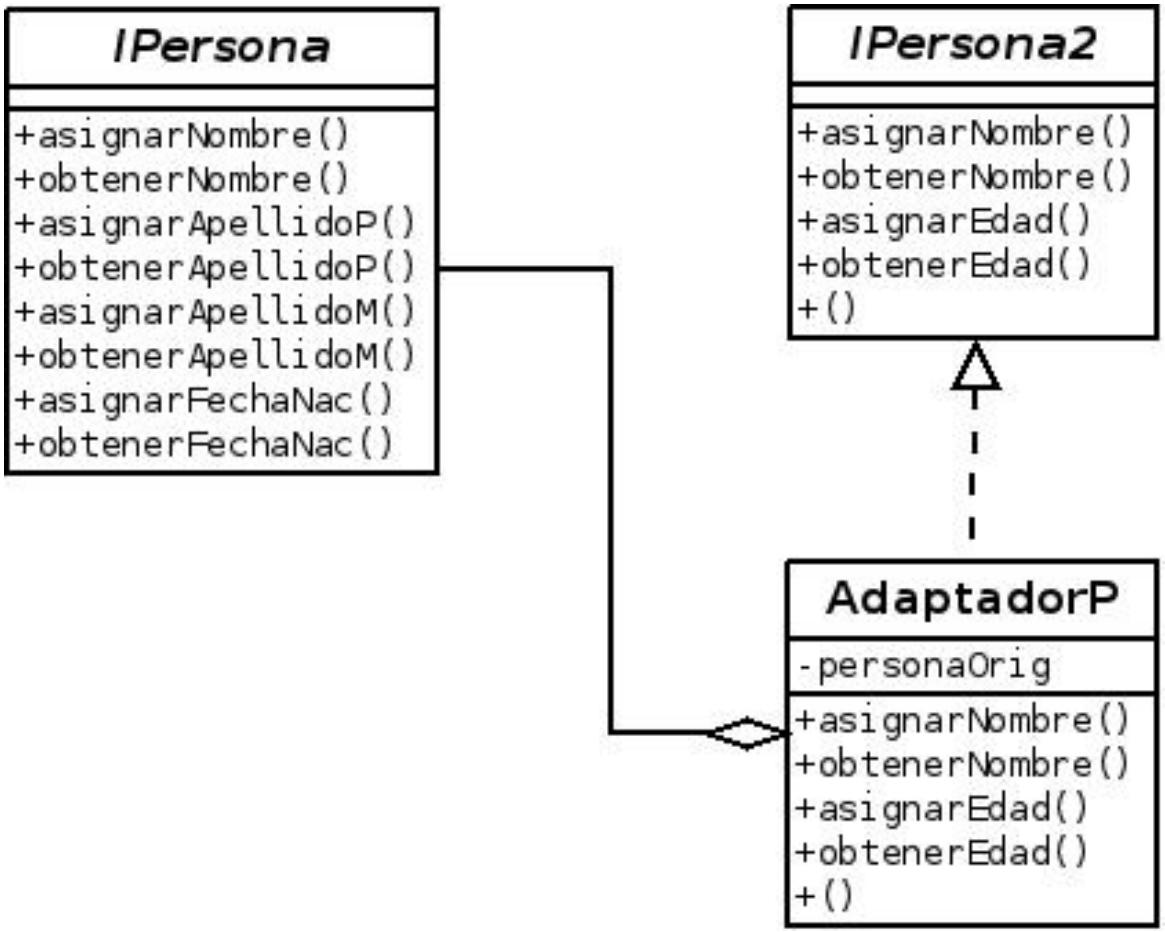




¿Qué hacer?

...Ejemplo

Crear un adaptador.



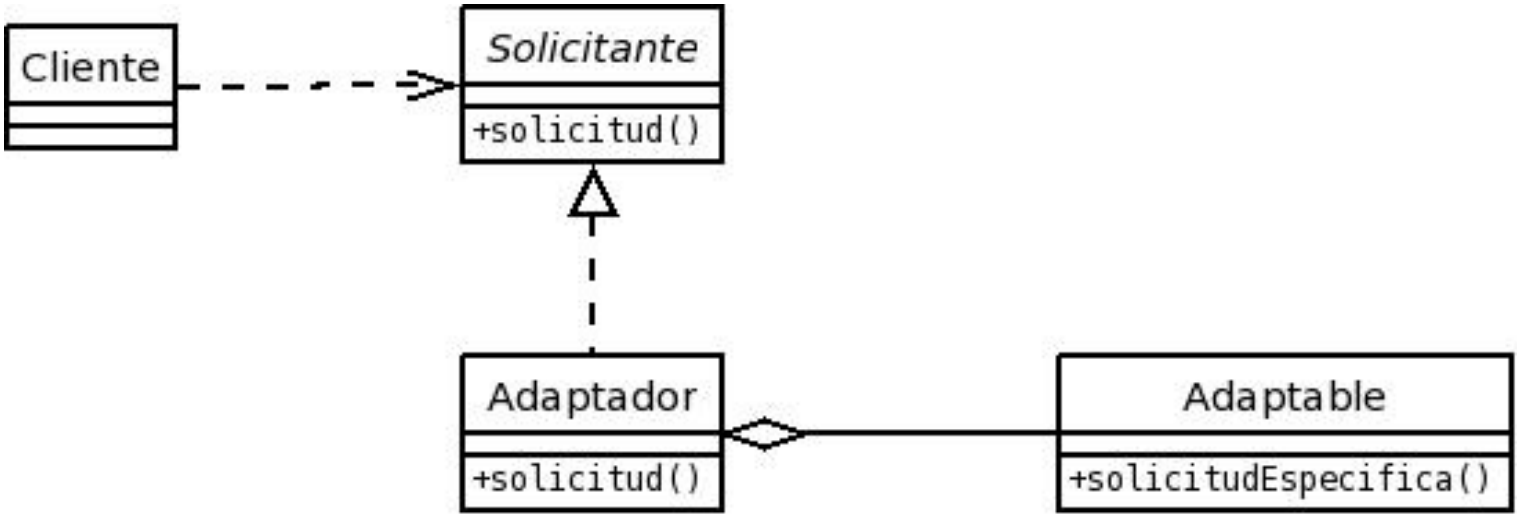
... Ejemplo (uso del adaptador)

```
public static void main(String[] pps) {
  PersonaOriginal original = new PersonaOriginal("Ana","Perez","Diaz
                                        new Date(2000,01,01));

  AdaptadorP nueva = AdaptadorP (original);

  System.out.println(nueva.obtenerEdad());
  System.out.println(nueva.obtenerNombre());

  nueva.asignarEdad(10);
  nueva.asignarNombre("Juan Sanchez Rosas");

  System.out.println(nueva.obtenerEdad());
  System.out.println(nueva.obtenerNombre());
}
```

Patrón adaptador

- Nombre: Adaptador
- Categoría: Estructural, objeto
- Objetivo: Actuar como intermediario entre dos clases, convirtiendo la interfaz de una clase para que pueda ser usada por otra.
  Un objeto `Adaptador` proporciona la funcionalidad prometida por una interfaz sin tener que conocer cuál clase es utilizada para implementarla. Permite trabajar juntas a dos clases con interfaces incompatibles.
- Aplicabilidad: El patrón adaptador se debería utilizar cuando:
  - Se quiere utilizar una clase que llame a un método a través de una interfaz, pero se busca utilizarlo con una clase que no implementa tal interfaz.
  - Se busca determinar dinámicamente que métodos de otros objetos llama un objeto.
  - No se quiere que el objeto llamado tenga conocimientos de la otra clase de objetos.

... Patrón adaptador (Diagrama UML)



Los participantes de este patrón son:

- `Cliente`: colabora con la conformación de objetos para la interfaz `Solicitante`.
- `Solicitante`: define la interfaz específica del dominio que usa `Cliente`.
- `Adaptable`: define una interfaz existente que necesita adaptarse
- `Adaptador`: adapta la interfaz de `Adaptable` a la interfaz `Solicitante`.

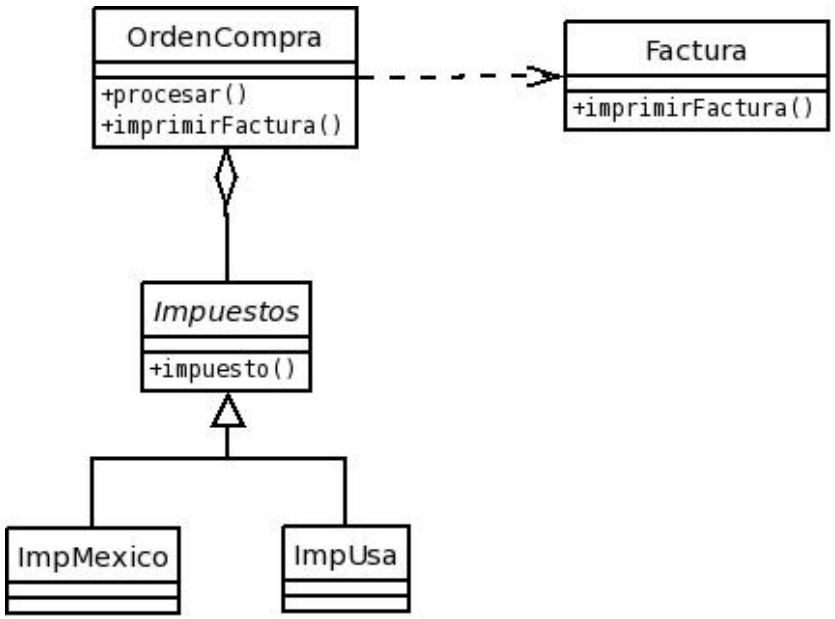
... Patrón adaptador

Consecuencias:

- El cliente y las clases adaptables permanecen independientes unas de las otras.
- Permite que un único adaptador trabaje con muchos adaptables. El adaptador también puede agregar funcionalidad a todos los adaptables de una sola vez.
- Puede hacer que un programa sea menos entendible.

Patrón decorador

El patrón decorador trata con el problema de agregar o eliminar funcionalidad a un objeto existente sin modificar su apariencia externa.
Suponer que se tiene un programa para imprimir recibos de compra/venta con el siguiente esquema:



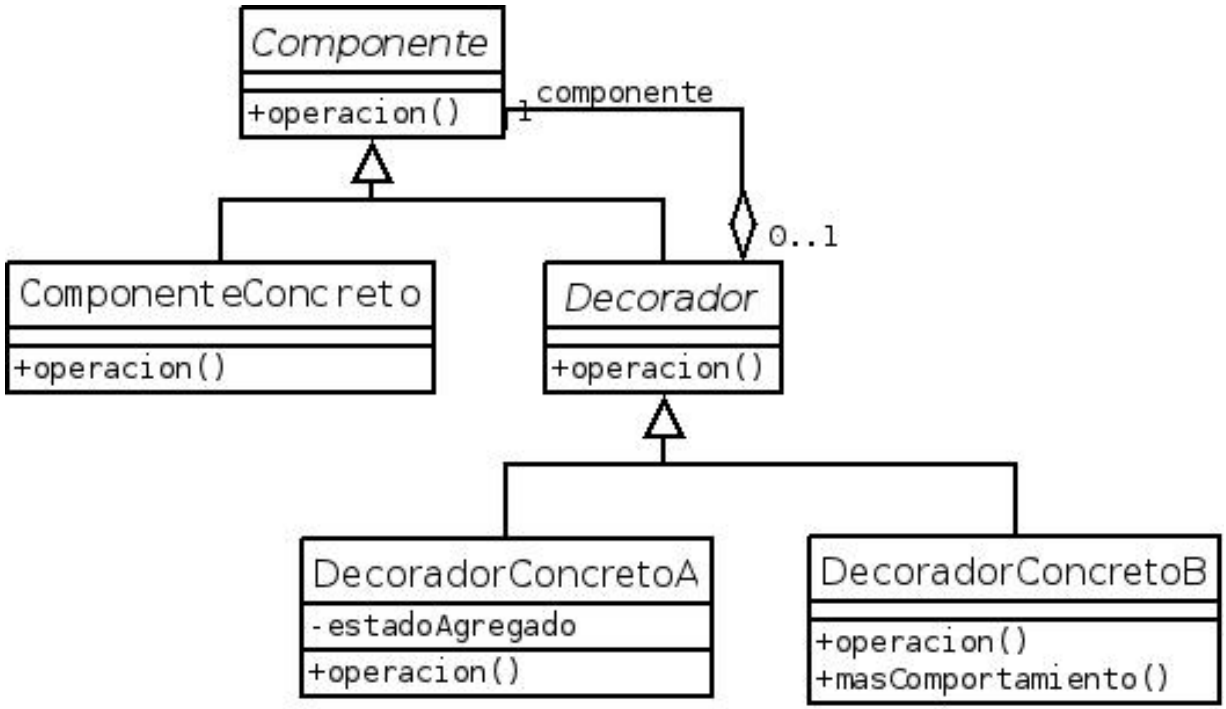
Ahora se requiere agregar información en el encabezado de una nota.
¿Qué hacer?
Crear una subclase `FacturaConEncabezado` que herede de `Factura`.

Si más adelante surge la necesidad de crear facturas con pie de página. Entonces se tienen las siguientes opciones:

- Crear otra subclase de `Nota`: `NotaConPie`.
- Crear una subclase de `NotaConEncabezado`: `NotaConEncabezadoYPie`.
- Crear clases para todas las combinaciones posibles de funcionalidades. `NotaConEncabezado`, `NotaConPie` y `NotaConEncabezadoYPie`; con tres funcionalidades tendríamos ocho clases y con cuatro, ¡dieciséis!

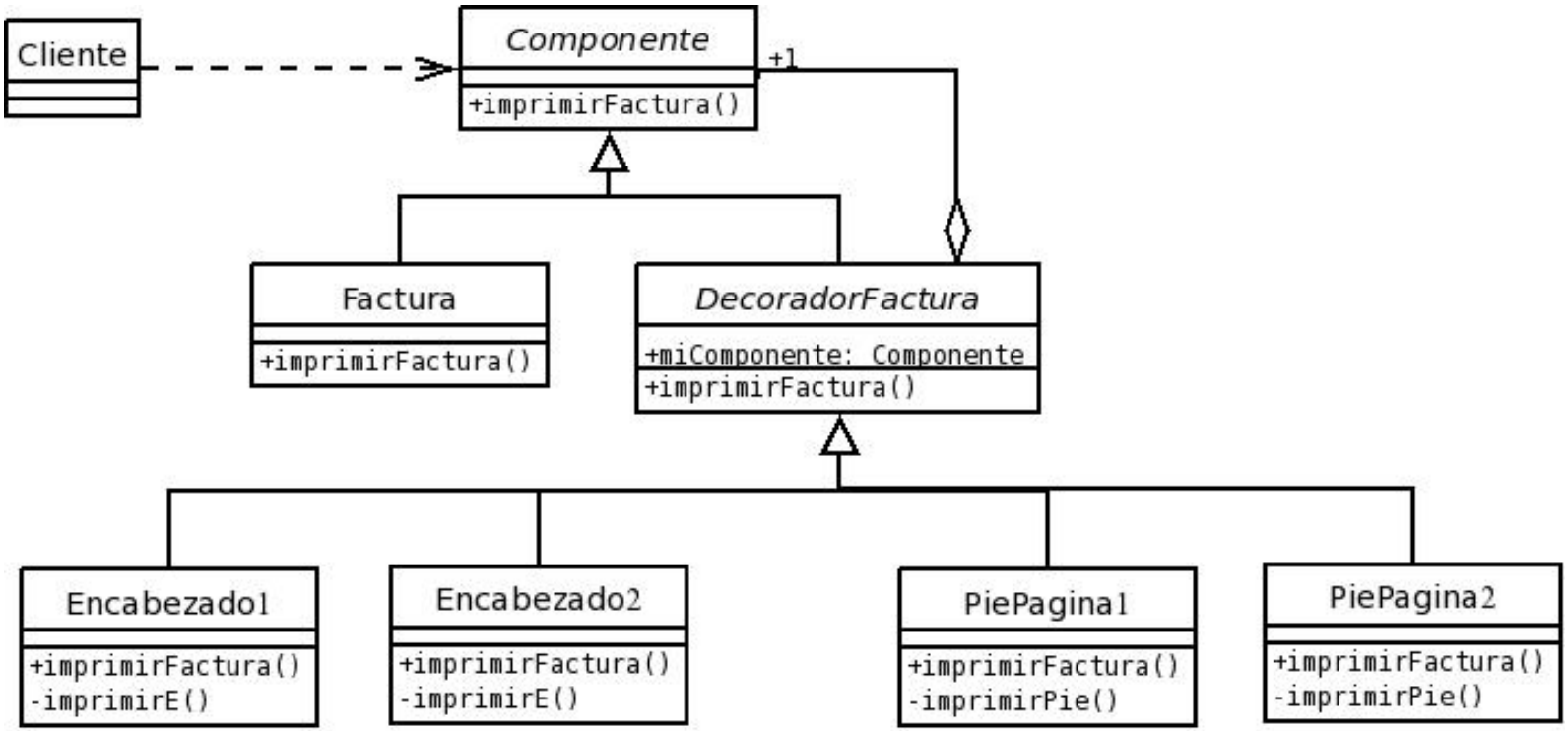
... Patrón decorador

- Propósito: Ligar, dinámicamente, responsabilidades o capacidades adicionales a un objeto. Los decoradores proporcionan una alternativa flexible a las subclases para extender funcionalidad.



- Participantes:
  - *Component* define la interfaz para los objetos que pueden tener responsabilidades agregadas dinámicamente.
  - *ConcreteComponent* define un objeto con responsabilidades adicionales.

... Patrón decorador (Solución)



... Patrón decorador (Solución)

```
public class Cliente {
  public static void main(String[] args) {
    Prueba miPrueba;
    miPrueba = new Prueba();
    Componente [] misComponentes = miPrueba.obtenerComponentes();
    for (int i=0; i < misComponentes.length; i++)
      misComponentes[i].imprimirNota();
  }
}
abstract public class Componente {
    abstract public void imprimirNota();
}
public class NotaDeVenta extends Componente {
    public void imprimirNota() {
       System.out.println("Imprime una nota de venta");
    }
}
```

... Patrón decorador (Solución)

```
abstract public class DecoradorDeNota extends Componente {
  private Componente miRestoDeNota;
  public DecoradorDeNota (Componente miComponente) {
    miRestoDeNota= miComponente;
  }
  public void RestoDeNota () {
    if (miRestoDeNota != null) miRestoDeNota.imprimirNota();
  }
}

public class Encabezado1 extends DecoradorDeNota {
    public Encabezado1 (Componente miComponente) {
        super(miComponente);
    }
    public void imprimirNota () {
        System.out.println("Imprime una nota de venta con encabezad
        super.RestoDeNota();
    }
```

... Patrón decorador (Solución)

```
public class PieDePagina1 extends DecoradorDeNota {
    public PieDePagina1 (Componente miComponente) {
        super(miComponente);
    }
    public void imprimirNota() {
        super.RestoDeNota();
        System.out.println("Imprime una nota de venta con pie de pag
    }
}

public class PieDePagina2 extends DecoradorDeNota {
    public PieDePagina2 (Componente miComponente) {
        super(miComponente);
    }
    public void imprimirNota() {
        super.RestoDeNota();
        System.out.println("Imprime una nota de venta con pie de pag
    }
```

... Patrón decorador (Solución)

```
public class Prueba {
    public Componente[] obtenerComponentes () {
        Componente[] misComponentes = new Componente[5];

        misComponentes[0]= new NotaDeVenta();
        misComponentes[1]= new PieDePagina1(misComponentes[0]);
        misComponentes[2]= new Encabezado1(misComponentes[0]);
        misComponentes[3]= new PieDePagina1(misComponentes[0]);
        misComponentes[4]= new Encabezado1(misComponentes[0]);

        return misComponentes;
    }
}
```

Patrón puente

- El patrón **puente** es un patrón estructural. Este permite separar una abstracción de su implementación de manera que ambas pueden variar independientemente.
- Cuando un objeto tiene unas implementaciones posibles, la manera habitual de implementación es el uso de herencia. Muchas veces la herencia se puede tornar inmanejable y, por otro lado, acopla el código cliente con una implementación concreta. Este patrón busca eliminar la inconveniencia de esta solución.

... Patrón puente (Ejemplo)

Suponer que se tiene una clase abstracta `Vehiculo` como sigue:

```
public abstract class Vehiculo {
  Motor motor;
  int pesoEnKilos;

  public abstract void conducir();

  public void setMotor(Motor motor) {
    this.motor = motor;
  }
  public void reporteDeVelocidad(int caballosDeFuerza) {
    int radio = pesoEnKilos / caballosDeFuerza;
    if (radio < 3) {
      System.out.println("El vehiculo va a alta velocidad.");
    } else if ((radio >= 3) && (radio < 8)) {
      System.out.println("El vehiculo va a velocidad promedio.");
    } else {
      System.out.println("El vehiculo va a baja velocidad.");
```

... Patrón puente (Ejemplo)

```
public class Autobus extends Vehiculo {

  public Autobus(Motor motor) {
    this.pesoEnKilos = 3000;
    this.motor = motor;
  }

  public void conducir() {
    System.out.println("\nEl autobus esta avanzando");
    int caballosDeFuerza = motor.avanzar();
    reporteDeVelocidad(caballosDeFuerza);
  }
}
```

... Patrón puente (Ejemplo)

```
public class Automovil extends Vehiculo {

  public Automovil(Motor motor) {
    this.pesoEnKilos = 600;
    this.motor = motor;
  }

  public void conducir() {
    System.out.println("\nEl automovil esta avanzando");
    int caballosDeFuerza = motor.avanzar();
    reportOnSpeed(caballosDeFuerza);
  }
}
```

... Patrón puente (Ejemplo)

La interfaz `Motor` declara el método `avanzar`.

```
public interface Motor {
  public int avanzar();
}

public class MotorGrande implements Motor {
  int caballosDeFuerza;

  public MotorGrande() {
    caballosDeFuerza = 350;
  }

  public int avanzar() {
    System.out.println("El motorsote esta funcionando");
    return caballosDeFuerza;
  }
}
```

... Patrón puente (Ejemplo)

```
public class MotorPequeño implements Motor {
  int caballosDeFuerza;

  public MotorPequeño() {
    caballosDeFuerza = 100;
  }

  public int avanzar() {
    System.out.println("El motorsito esta funcionando");
    return caballosDeFuerza;
  }
}
```

... Patrón puente (Ejemplo)

```
public class DemoPuente {
  public static void main(String[] args) {
    Vehiculo vehiculo = new Autobus(new MotorPequeño());
    vehiculo.conducir();
    vehiculo.setMotor(new MotorGrande());
    vehiculo.conducir();

    vehiculo = new Automovil(new MotorPequeño());
    vehiculo.conducir();
    vehiculo.setMotor(new MotorGrande());
    vehiculo.conducir();
  }
}
```

```
El autobus esta avanzando
El motorsito esta funcionando
El vehiculo va a baja velocidad.

El autobus esta avanzando
```

... Patrón Puente (Consecuencias)

- Desacopla interfaz e implementación. Permite a un objeto cambiar su implementación en tiempo de ejecución. Este desacoplamiento fomenta las capas, que pueden conducir a un sistema mejor estructurado.
- La parte de alto nivel de un sistema sólo tiene que conocer `Abstracción` e `Implementor`.
- Mejora la extensibilidad: se puede extender las jerarquías de `Abstraccion` e `Implementor` independientemente.
- Esconde los detalles de la implementación a los clientes

Patrón fachada

El patrón fachada busca simplificar el sistema, desde el punto de vista del cliente, proporcionando una interfaz unificada para un conjunto de subsistemas, definiendo una interfaz de nivel más alto. Esto hace que el sistema sea más fácil de usar.

El patrón Fachada facilita el uso de sistemas complejos, ya sea usando sólo un subconjunto del sistema o bien usandolo de una forma particular.

La mayoría del trabajo se realiza por el sistema subyacente. La fachada proporciona una colección de métodos más fáciles de entender. Ésatos a su vez utilizan el sistema subyacente para implementar las uevas funciones definidas.

... Patrón fachada

- Nombre: Fachada.
- Categoría: Patrón de diseño de comportamiento.
- Objetivo: Simplificar el uso de un sistema existente, a través de una interfaz propia.
- Aplicabilidad: Se debe utilizar cuando:
  - Se quiera proporcionar una interfaz sencilla para un subsistema complejo.
  - Se quiera desacoplar un subsistema de sus clientes y de otros subsistemas, haciéndolo más independiente y portable.
  - Se quiera dividir los sistemas en niveles: las fachadas serían el punto de entrada a cada nivel. Este patrón puede ser utilizado a nivel aplicación.

... Patrón fachada

Suponer que se está desarrollando un sistema para una inmobiliaria.
La inmobiliaria realiza diferentes trabajos como: cobrar alquiler, mostrar inmuebles, administrar consorcios, hacer contratos de ventas, contratos de alquiler, etc.
Suponer que en la inmobiliaria tenemos diversos tipos de Personas, todas con sus atributos y métodos correspondientes. (Clientes, Interesados, Propietarios)
Además de esas subclases de persona tiene las siguientes clases:

... Patrón fachada

```
public class AdministradorAlquiler {
  public void cobro (double monto) { ... }
  ...
}

public class CuentasAPagar {
  public void pagoPropietario (double monto) { ... }
  ...
}
public class MuestraPropiedad {
  public void mostrarPropiedad (id numeroPropiedad) { ... }
  ...
}

public class VentaInmueble {
  public void cerrarVenta () { ... }
  ...
}
```

... Patrón fachada

Para trabajar todos esos "sistemas" se crea una fachada:

```
public class Inmobiliaria {
  private MuestraPropiedad muestra;
  private VentaInmueble venta;
  private CuentasAPagar cuentas;
  private AdministracionAlquiler alquiler;

  public Inmobiaria() {
    muestra = new MuestraPropiedad();
    venta = new VentaInmueble();
    cuentas = new CuentasAPagar(9;
    alquiler = new AdministracionAlquiler();
  }

  public void atencionCliente(Cliente c) {
    System.out.println("Atendiendo a un cliente");
  }
```

... Patrón fachada

```
public void atencionPropietario(Propietario p) {
  System.out.println("Atendiendo a un propietario");
}

public void atencionInteresado(Interesado i) {
  System.out.println("Atendiendo a un interesado en una propiedad
}

public void atencion(Persona p) {
  if (p instanceof Cliente) {
    atencionCliente((Cliente) p);
  } else if (p instanceof Propietario) p) {
    atencionPropietario((Propietario) p);
  } else if (p instanceof Interesado) p) {
    atencionInteresado((Interesado) p);
```

```
public static void main(String[] pps) {
  Cliente c = new Cliente();    Interesado i = new Interesado();

  Inmobiliaria inmo1 = Inmobiliaria();   // Cliente 1
  inmo1.atencionCliente (c);
  inmo1.atencionInteresado (i);
  MuestraPropiedad muestra = new MuestraPropiedad();
  muestra.mostrarPropiedad(123);
  VentaInmueble venta = new VentaInmueble();
  venta.cerrarVenta();
  AdministracionAlquiler alquiler = new AdministracionAlquiler();
  alquiler.cobrar(1200);
  CuentasAPagar cuentas = new CuentasAPagar();
  cuentas.pagarPropietario(1.8);

  Inmobiliaria inmo2 = Inmobiliaria();   // Cliente 2 que usa la fa
  inmo2.atencion(i);
  inmo2.atencion(c);
  inmo2.mostrarPropiedad(123);
  inmo2.cerrarVenta();
  inmo2.cobrar(1200);
```

Patrones de creación

Facilitan la tarea de crear objetos;

- Permitiendo crear objetos sin tener que identificar una clase específica.
- Evitando escribir código grande o complejo para crear objetos.
- Forzando restricciones en el tipo o el número de objetos que pueden ser creados.

Los patrones de creación son:

- `Abstract Factory`: Proporciona una interfaz para crear familias de objetos relacionados o dependientes sin especificar su clase concreta.
- `Builder`: Permite construir un objeto complejo especificando sólo su tipo y contenido.
- `Factory Method`: Define una interfaz para crear un objeto dejando a las subclases decidir el tipo específico al que pertenecen.
- `Prototype`: Permite crear objetos personalizados sin conocer su clase exacta a los detalles de cómo crearlos.
- `Singleton`: Garantiza que solamente se crea una instancia de la clase y provee un punto de acceso global a él.

Factory method (Método de fabricación)

Escribir un traductor que devuelva los números del cero al diez en tres idiomas: inglés, español, y alemán.

```
public class Cliente {
  public Cliente(){}

  public String traducirNumero(String idioma, int numero){
    if (idioma.equals("español")){
      switch (numero){
         case 1: return "uno";
         case 2: return "dos";
       //     ....
       }
    }
     if (idioma.equals("english")){
        switch (numero){
         case 1: return "one";
         case 2: return "two";
       //         ....
```

```
    if (idioma.equals("deutsch")){
       switch (numero){
        case 1: return "eins";
        case 2: return "zwei";
      //         ....
        }
    }
    return "No hablo " + idioma;
  }//traducirNumero

  public static void main(String args[]){
    Cliente cte = new Cliente();
    System.out.println(cte.traducirNumero("espanol",1));
  }
}
```

Modificaciones

Se solicita la traducción de cualquier número ¿?
Definir una clase abstracta `Traductor`, y para cada idioma crear una subclase de ella.

```
public abstract class Traductor{  // Interfaz creador
    public abstract String traducirNumero(int numero);
}

public class TraductorEspañol extends Traductor {
   public TraductorEspañol(){
    super();
    ...
  }
   public String traducirNumero(int numero){
    switch(numero){
      case 1: return "uno";
      case 2: return "dos";
          ...
    }
```

La clase para el inglés iría

```
public class TraductorIngles extends Traductor {

  public TraductorIngles(){
    super();
    ...
  }

  public String traducirNumero(int numero){
    switch(numero){
      case 1: return "one";
      case 2: return "two";
          ...
    }
  }
}

Traductor t = new TraductorEspañol();
t.traducirNumero(1);
```

La clase Cliente cambiaría:

```
public class Cliente {

  public String traducirNumero(String idioma){
    Traductor traductor = "No hablo " + idioma;
    if (idioma.equals("español"))
      traductor = new TraductorEspañol();
    else
    if (idioma.equals("ingles"))
      traductor = new TraductorIngles();
    else
    if (idioma.equals("aleman"))
      traductor = new TraductorAleman();

    return traductor;
  }

  public static void main(String args[]){
    Cliente cte = new Cliente();
    System.out.println(cte.traducirNumero("español",1));
  }
```

Se ha ganado :

1. El código es más legible.
2. El código es más escalable. Es posible agregar el traductor para el francés muy fácilmente.
3. Se ha ocultado la manera en la que se traduce.

¿Qué no está tan bien?

- `Cliente` está haciendo algo que no le compete: está eligiendo la instancia de `Traductor` que quiere usar.
- Si se usara el traductor en 100 lugares, en 100 lugares se tiene que buscar qué clase de `Traductor` instanciar.

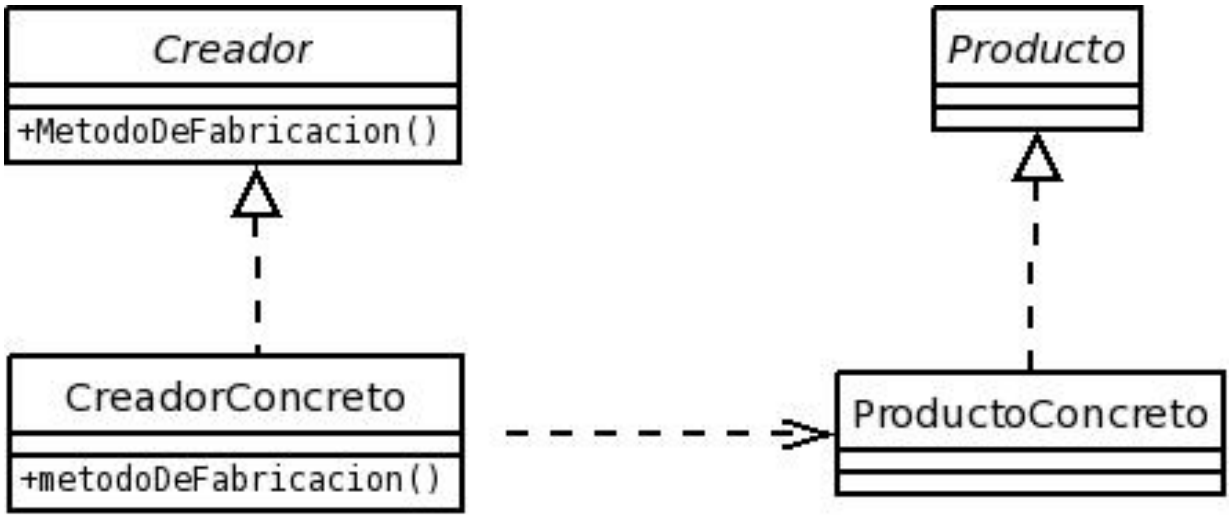
El patrón de fabricación *método de fabricación* esconde esa lógica.

Patrón Método de fabricación

- Nombre: Método de fabricación
- Categoría: Creación
- Propósito: Definir una interfaz para crear un objeto pero diferir la creación a las subclases.
- También conocido como: constructor virtual.
- Uso: Usar el patrón Método de fabricación cuando:
  - una clase no pueda anticipar la clase de los objetos que debe crear.
  - una clase difiere a sus subclases especificar los objetos que va a crear.

... Patrón Método de fabricación

- Diagrama:



- Participantes:
  - `Product` define la interfaz de los objetos que serán creados.
  - `ConcretProduct` implementa la interfaz `Producto`.
  - `Creator` define uno o más métodos de fábrica que crean los productos abstractos, esto es, objetos del tipo `Producto`.
  - `ConcretCreator` sobreescribe el método de fabricación para regresar una instancia de un producto concreto.

Uso del patrón Método de fabricación

```
public class TraductorFactory {    // Creador concreto

  public TraductorFactory(){ }

  public static Traductor crearTraductor(String idioma){
    Traductor traductor = null;
    if (idioma=="español")
      traductor = new TraductorEspañol();
    else
    if (idioma=="english"){
      traductor = new TraductorIngles();
     else
    if (idioma=="deutsch")
      traductor = new TraductorAleman();

    return traductor;
  }
}
```

¿Qué hace `TraductorFactory`?
Elige qué clase de traductor se instanciará.
El `Cliente` de nuevo cambia y quedaría así:

```
public class Cliente {
  public static void main(Strin []args){
    Traductor traductor = TraductorFactory.crearTraductor("español")
    System.out.println( traductor.traducirNumero(1) );
  }
}
```

Abstract Factory (Fabricación abstracta)

Hacer un reloj que muestra la hora actual. La hora puede ser desplegada en formato de 24Hrs o en formato AM/PM.

```
public abstract class Reloj {
  abstract String dameLaHora();
}
public class RelojAmPm extends Reloj{
  public RelojAmPm(){}

  public String dameLaHora() {
    Date d = new Date();
    int hora = d.getHours();
    int minutos = d.getMinutes();
    int segundos = d.getSeconds();

    return (hora<=12)
           ? "Son las "+hora+":"+minutos+":"+segundos+" AM";
           : "Son las "+(hora-12)+":"+minutos+":"+segundos+" PM";
  }
```

```
public class Reloj24Hrs extends Reloj {
  public String dameLaHora() {
    Date d = new Date();
    int hora = d.getHours();
    int minutos = d.getMinutes();
    int segundos = d.getSeconds();

    return "Son las " + hora + ":" + minutos + ":" + segundos + " "
   }
}
```

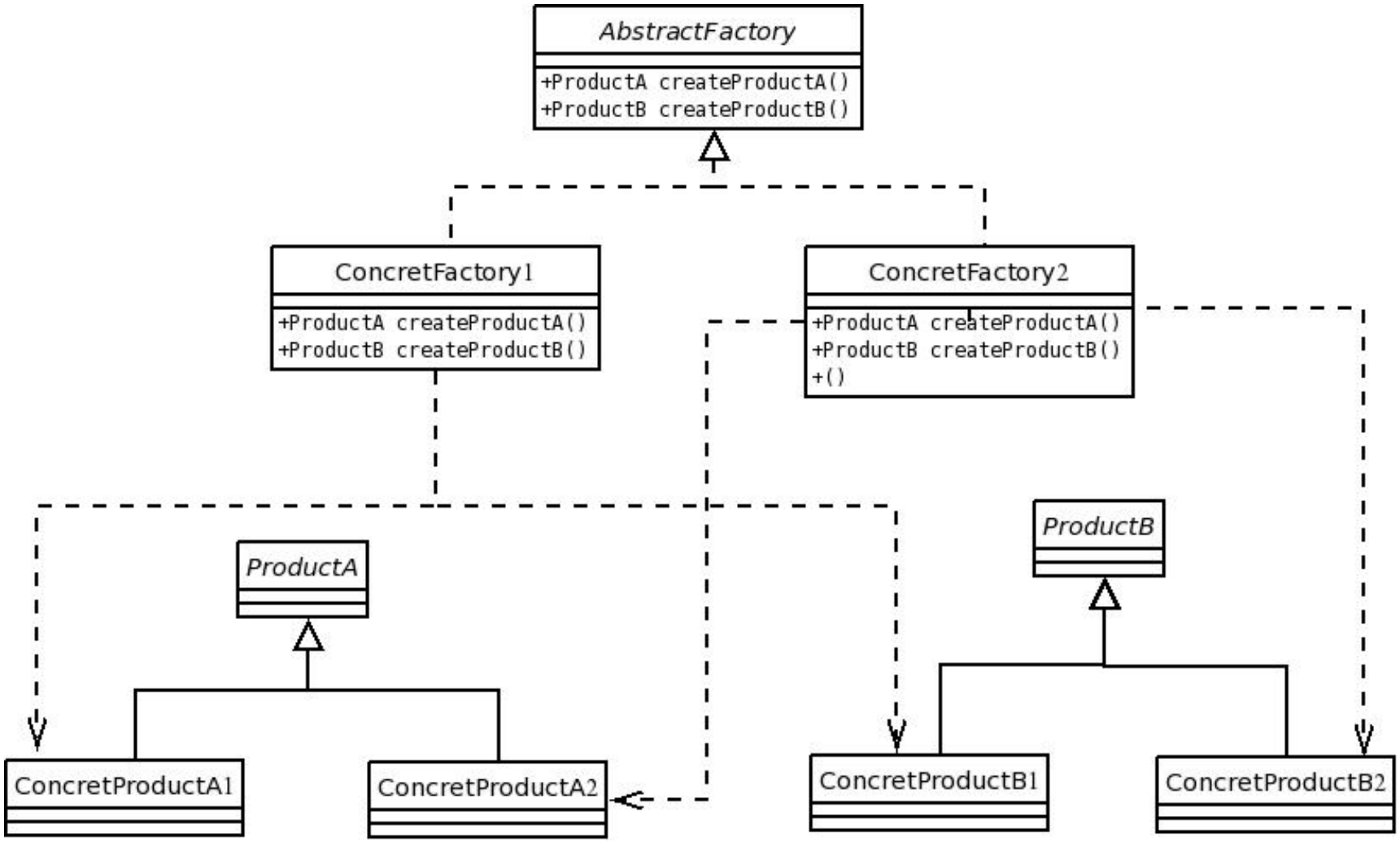
```
public class RelojFactory {
  public static final int RELOJ_AM_PM=0;
  public static final int RELOJ_24_HRS=1;

  public RelojFactory(){}

  public static Reloj crearReloj(int tipoDeReloj){
    if (tipoDeReloj==RelojFactory.RELOJ_24_HRS)
      return new Reloj24Hrs();
    if (tipoDeReloj==RelojFactory.RELOJ_AM_PM)
      return new RelojAmPm();
    return null;
  }
}

public class Cliente {
  public static void main(String[] args) {
    Reloj r = RelojFactory.crearReloj(RelojFactory.RELOJ_24_HRS);
    System.out.println(r.dameLaHora());
  }
}
```

Patrón: Fabricación abstracta

- Nombre: Fábrica abstracta
- Categoría: Patrón de creación
- Propósito: Proporcionar una interfaz para crear una familia de objetos relacionados y dependientes sin especificar su clase concreta.
- Uso: Usar el patrón de diseño fábrica abstracta cuando:
  - un sistema debería ser independiente de cómo se crean sus componentes o productos.
  - un sistema debería ser configurable con una de varias familias intercambiables de productos.
  - una familia de productos relacionados no debería mezclarse con productos similares de familias diferentes.
  - sólo las interfazs de los productos se exponen, en tanto la implementación de los productos no se revela.

...Patrón: Fabricación abstracta

- Diagrama:



- Participantes:
  - *AbstractFactory* define métodos para crear productos abstractos y

Uso de Abstract Factory

```
public abstract class FabricacionAbs {
  public static final String US="ESTADOS_UNIDOS";
  public static final String MX="MÉXICO";

  String pais;

  public abstract Traductor crearTraductor();
  public abstract Reloj crearReloj();

  public String getPais(){
    return this.pais;
  }

  public void setPais(String pais){
    this.pais = pais;
  }
}
```

```
public class FabricaMexico extends FabricacionAbs{
   public FabricaMexico(){
    this.pais=this.MX;
  }
  public Traductor crearTraductor() {
    return FabricaTraductores.crearTraductor("español");
  }
  public Reloj crearReloj() {
    return RelojFactory.crearReloj(RelojFactory.RELOJ_24_HRS);
  }
}

public class FabricaUSA extends FabricacionAbs{
  public FabricaUSA(){
    this.pais=FabricacionAbs.US;
  }
  public Traductor crearTraductor() {
    return FabricaTraductores.crearTraductor("ingles");
  }
  public Reloj crearReloj() {
    return RelojFactory.crearReloj(RelojFactory.RELOJ_AM_PM);
```

```
public class Cliente {
  public static void main(String[] args) {
    Reloj reloj = null;
    Traductor traductor = null;

    FabricacionAbs localeFactory = new FabricaMexico();
    reloj = localeFactory.crearReloj();
    traductor = localeFactory.crearTraductor();

    System.out.println("--------");
    System.out.println("1="+traductor.traducirNumero(1));
    System.out.println(reloj.dameLaHora());
  }
}
```

Resultado: ?

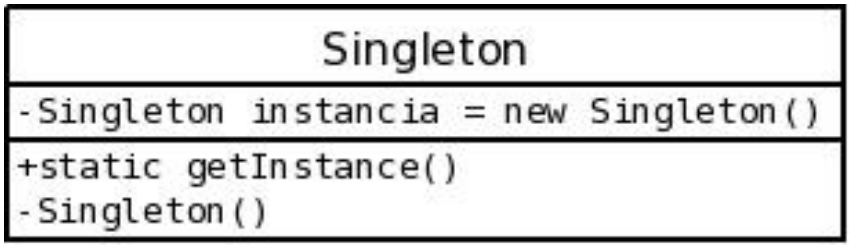
Al cambiar `FabricacionAbs localeFactory = new FabricaMexico();` por:

`FabricacionAbs localeFactory = new FabricaUSA();`

Resultado = ?

Patrón Singleton

El patrón Singleton, también es un patrón de creación pero a diferencia de los dos anteriores, no se encarga de la creación de objetos en sí, sino que se enfoca en la restricción en la creación de un solo objeto.



Ejemplo de uso

Se ha utilizado un Traductor.

```
public class Traductor{
    private static boolean instanciado=false;
    private static Traductor traductor;

    private Traductor(){
      //cargar un diccionario a memoria  a través de un WebService.
   }
   public static Traductor getTraductor(){
       if (! Traductor.instanciado){
         Traductor.traductor= new Traductor();
         Traductor.instanciado=true;
       }
      return Traductor.traductor;
   }
   public String traducir(String duda){
     //... código aquí
  }
```

Para utilizar el traductor:

```
Traductor.getTraductor().traducir(ünaPalabra");
```

Con lo cual se logra que no se pueda hacer esto

```
Traductor t = new Traductor();
```

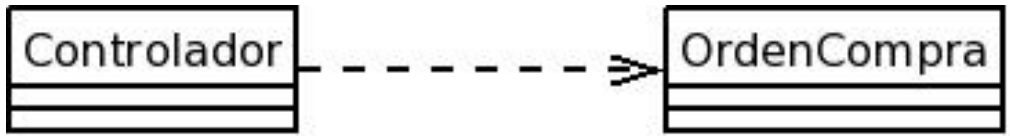
Otra forma de hacerlo

```
public class Traductor{
  private static  Traductor traductor=null;
  private Traductor(){
    //cargar un diccionario a memoria  a través de un WebService.
  }
  public static Traductor getTraductor(){
    if (traductor==null)
      traductor= new Traductor();
    return traductor;
  }
  public String traducir(String duda){
     // ...
  }
}
```

Programa ejemplo

Patrón Estrategia

Suponer que se tiene un sistema que permite compras en línea en México.



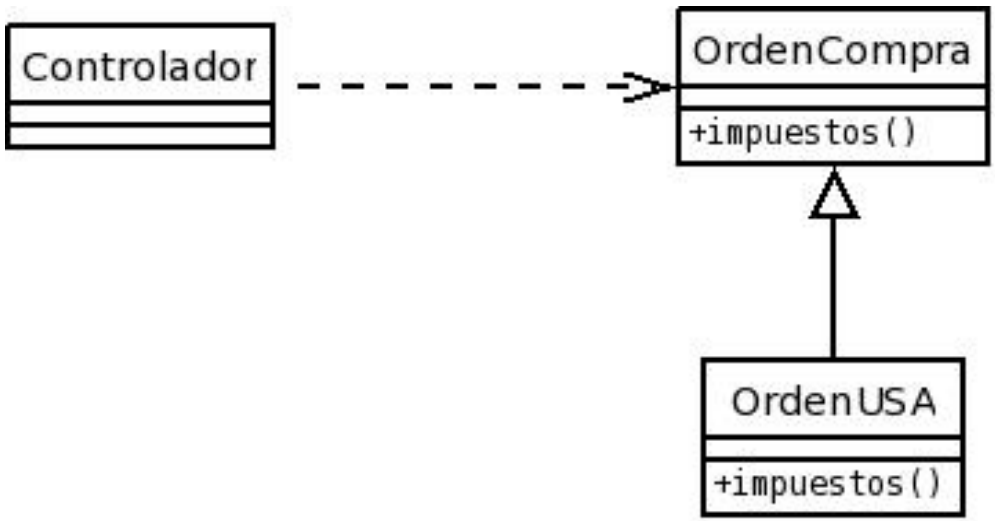
Las funciones de `OrdenDeCompra` incluyen:

- Permitir llenar la orden desde una GUI.
- Manejar el cálculo de los impuestos.
- Procesar la orden, imprimiendo una factura.

Algunas de estas funciones serán implementadas con la ayuda de otros objetos.

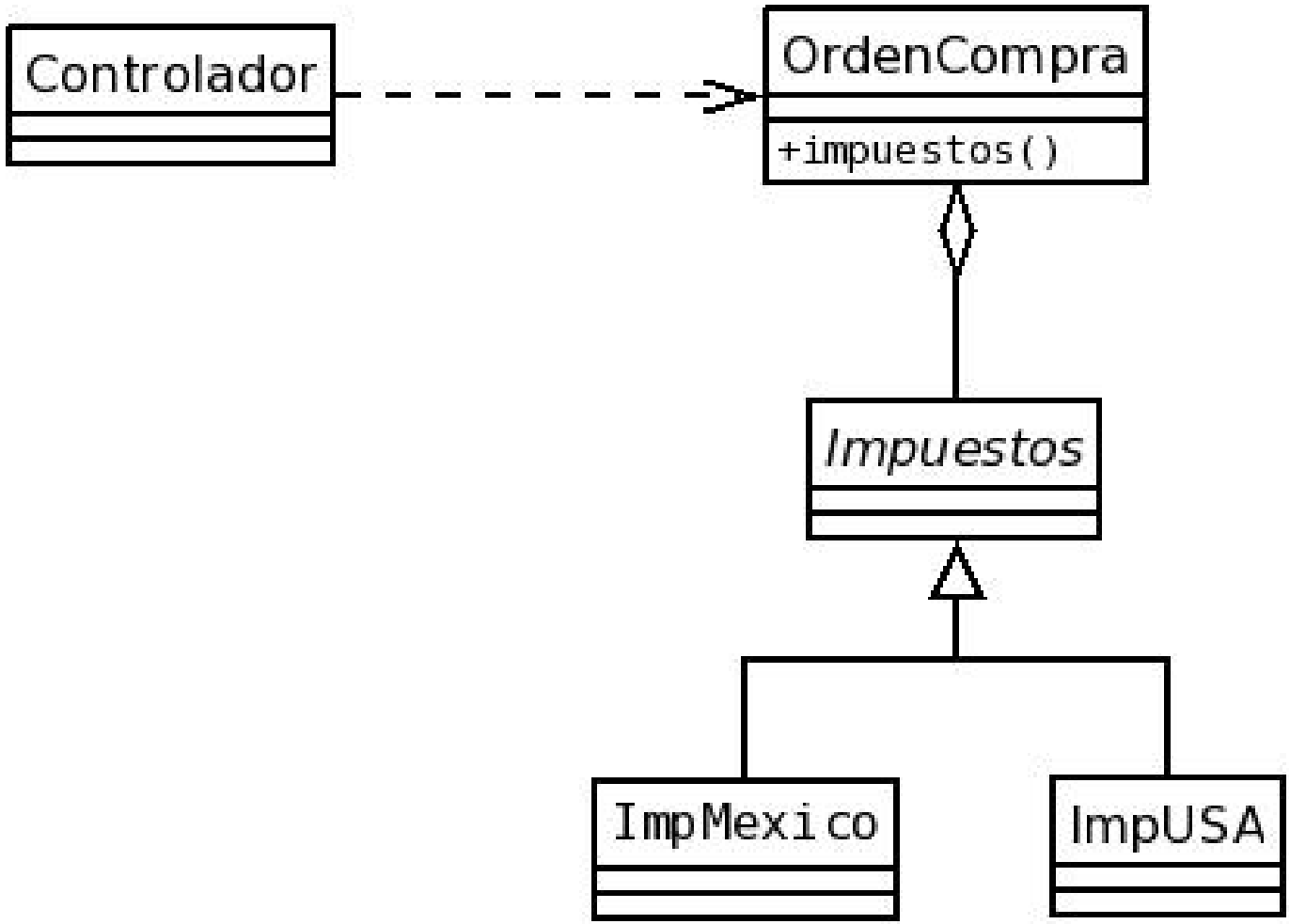
... Patrón Estrategia

Una vez funcionando la aplicación se cambia la forma de calcular los impuestos, pues la empresa se ha expandido a otros países.



- Problema: la jerarquía que se está construyendo no será fácilmente manejable.
- La especialización repetida tal como está causará que el código resulte no entendible o redundante.
- Otro enfoque sería "encontrar la fuente de posible variación, encapsularla y favorecer la agregación más que la herencia".
  En el ejemplo, se ha identificado que las reglas para los impuestos varían.
- Crear un objeto `Impuesto` que defina la interfaz para lograr esta tarea. Luego crear las subclases especificas necesarias.
  Figura ...

Ahora en lugar de hacer diferentes versiones de las ordenes de compra usando herencia, trabajar la variación con la composición.



Ventajas:

- Se ha mejorado la cohesión al tener una clase impuestos.
- Si se tiene un nuevo requerimiento, simplemente se necesita derivar una nueva clase de `Impuesto` que lo implemente.
- Permite que las reglas de negocio varíen con independencia del objeto `OrdenDeCompra` que lo usa.

Esencialmente encapsular un algoritmo en una clase abstracta y después usarlo de manera intercambiable es el patrón estrategia.

Patrón de diseño Estrategia