# 1. Introducción.

Notes & Todo: Términos en inglés y en español. Usando ejemplos. Ejemplos propios. They facilitate communication

# 2. Los patrones de diseño

El software es un tema muy amplio que tiene bastantes esquinas interesantes para digerir. Uno de los puntos cruciales de todo el ecosistema técnico y de negocios del software es la calidad del código. Crear código de alta calidad no es fácil, y para cualquier profesional en el área, los fundamentos de hacer buen software siempre deben ser revisados constantemente. Porque, aun con la avalancha de nuevos lenguajes de programación, paradigmas y *frameworks*, los fundamentos de hacer buen software son similares.

En el caso específico de la programación orientada a objetos, existen desde hace un par de décadas los llamados patrones de diseño. Estos patrones, que usualmente aparecen como consecuencia de identificar anti-patrones.

En sí, un patrón de diseño es una solución puntual a un problema común y repetido de diseño. Para que sea considerado como tal, debe tener una amplia cantidad de escenarios donde resuelva estos problemas.

La solidificación del concepto se dio en 1994, cuando Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides (conocidos como "the gang of four (GoF)", o “la mara de los cuatro”) publicó el famoso libro “Design Patterns: Elements of Reusable Object-Oriented Software” (Patrones de diseño: elementos reusables para software orientado a objetos). A partir de ahí el concepto se popularizó, y las herramientas de desarrollo comienzan a incorporar el uso de patrones por todos lados, apareciendo el concepto de “refactorizar”, que es el concepto de aplicar patrones de diseño a software ya existente.

# 3. Patrones Creacionales.

Los patrones creacionales son los primeros que se abordarán, ya que naturalmente son los que aparecen inicialmente en las distintas etapas de desarrollo.

La creación de objetos en un lenguaje de programación orientada a objetos es percibida como algo muy simple y fundamental al paradigma. Si bien lo es, conforme un sistema comienza a escalar, comienza también a volverse muy importante la manera en la que los objetos son creados, afectando la manera que se hace la escalabilidad y el rendimiento.

Aunque la mayoría de máquinas virtuales y plataformas actuales cuentan con compiladores muy capaces que se enfocan en técnicas agresivas de optimización, nada puede sustituir a un software bien diseñado y bien hecho. En un sistema cualquiera, la cantidad de objetos que se crean y destruyen es considerable (independientemente del manejo de memoria que se elija), por lo que crear y destruir objetos de una buena manera es siempre un reto.

Dejando el rendimiento por un lado, una de las características fundamentales de un software que pueda escalar es que tan legible para humanos es. Ya se habló de lo eficiente que son los compiladores, por lo que enfocarse en hacer software legible es siempre una buena apuesta en cualquier proyecto de software. Y los patrones creacionales son muy útiles para esto.

La razón de ser de estos patrones es para facilitar, ordenar, o ayudar en la creación de objetos. Dependiendo del lenguaje de programación que estemos utilizando, generalmente crearemos un objeto así: Objeto o = new Objeto(). Los patrones creacionales modifican un poco la manera de crear objetos, y una manera de crearlos, por ejemplo, sería: Objeto o = ObjectoFactory.getInstance("x").

## 3.1 Patron de constructor fluido (*Builder, Fluid*)

En términos generales un constructor (*builder*) existe para esconder los detalles de la creación de un objeto final, al que se llama producto. Hay varios métodos para lograr esto, y por ello hay varias "implementaciones" de este patrón que en nada coinciden, salvo en el nombre. El clásico, usado en el GoF es un poco más complejo del que se presentará ahora. El enfoque estará en lo que se conoce como constructores de interfaces fluidas.

Para ilustrar el uso del patrón, se utilizará un ejemplo. Supóngase que se va a hacer un programa para una empresa que ofrece servicios de* web hosting* (alojamiento en web). Esta empresa comercializa varios planes: un plan personal, un plan bronce, uno plata, uno oro y finalmente uno diamante. El plan personal ofrece un alojamiento de 10 MB, una transferencia mensual de 100 MB, una cuenta de correo electrónico y nada más. El bronce ofrece 100MB de alojamiento, 1000 MB de transferencia mensual, y 10 cuentas de correo electrónico. Así sucesivamente cada plan aumenta las cantidades de alojamiento, transferencia y cuentas de correo. A la vez cada plan agrega nuevas características: el plata ofrece acceso ssh, y una base de datos; el oro agrega estadísticas de sitio y panel de control. Así se van multiplicando las opciones, y para crear la aplicación se define un objeto como el siguiente:

```java
package com.guisho.software.patrones.builder;
import java.math.BigDecimal;
public class PaqueteDeHosting {

    /*Los siguientes campos son obligatorios siempre*/
    private String nombre;
    private BigDecimal precioAnual;
    private int capacidadDeAlmacenamiento; //en MB
    private int transferenciaMensual; //en MB
    private int cantidadDireccionesCorreo;//
    /*Las siguientes son opcionales, hay planes que no los tienen*/
    private int cantidadSitiosPermitidos;
    private int cantidadBaseDeDatos;
    private String codigoOferta;
    private boolean accesoSsh;
    private boolean panelDeControl;
    private boolean estadisticasDeSitio;
    private boolean ipPublica;
    public PaqueteDeHosting(){
}

    /* mas constructores */

    /* Setters, getters y demás código....*/

}
```

Bien, ahora cada plan tiene una configuración previamente establecida, que el vendedor no arma en el momento, y que preferiblemente no puede cambiar. ¿Cómo se puede hacer para crear cada objeto? La primera manera que se suele proponer es crear un constructor para cada caso, manteniendo siempre los valores obligatorios. Esto resultaría en una colección de constructores como la siguiente:

Java

```java
    public PaqueteDeHosting(String nombre, BigDecimal precioAnual, int almacenamie

        this.nombre = nombre;
        this.precioAnual = precioAnual;
        this.capacidadDeAlmacenamiento = almacenamiento;
        this.transferenciaMensual = transferencia;
        this.cantidadDireccionesCorreo = cantidadCorreos;

    }

    public PaqueteDeHosting(String nombre, BigDecimal precioAnual, int almacenamie

        this.nombre = nombre;
        this.precioAnual = precioAnual;
        this.capacidadDeAlmacenamiento = almacenamiento;
        this.transferenciaMensual = transferencia;
        this.cantidadDireccionesCorreo = cantidadCorreos;
        this.cantidadBaseDeDatos = basesDatos;

    }

    public PaqueteDeHosting(String nombre, BigDecimal precioAnual, int almacenamie

        this.nombre = nombre;
        this.precioAnual = precioAnual;
        this.capacidadDeAlmacenamiento = almacenamiento;
        this.transferenciaMensual = transferencia;
        this.cantidadDireccionesCorreo = cantidadCorreos;
        this.ipPublica = ipPublica;

    }

    public PaqueteDeHosting(String nombre, BigDecimal precioAnual, int almacenamie

        this.nombre=nombre;
        this.precioAnual=precioAnual;
        this.capacidadDeAlmacenamiento=almacenamiento;
        this.transferenciaMensual=transferencia;
        this.cantidadDireccionesCorreo=cantidadCorreos;
        this.ipPublica=ipPublica;
        this.cantidadBaseDeDatos=basesDatos;

    }
```

Como se observa, sólo se hicieron un par de combinaciones con la cantidad de bases de datos y la IP pública. Mientras la cantidad de campos opcionales crece, la cantidad de constructores aumenta desmedidamente creando el ambiente ideal para que aparezcan errores.

Otro camino que se puede tomar es el clásico *bean*: un constructor vacío y *setters* para cada parámetro que se desee agregar. Sin embargo, este método tiene un pequeño inconveniente: puede dejar al objeto en un estado inconsistente: podría ponérsele cuántas cuentas de correo pero no ponérsele nombre, ni ponerle precio. ¿Qué se puede hacer entonces? Utilizar el patrón constructor y hacer un *builder*. El *builder* se explicará por el solo. Véase:

En PaqueteDeHosting se hace un constructor con los campos que siempre van para evitar estados inconsistentes:

```java
    public PaqueteDeHosting(String nombre, BigDecimal precioAnual, int almacenamie

        this.nombre = nombre;
        this.precioAnual = precioAnual;
        this.capacidadDeAlmacenamiento = almacenamiento;
        this.transferenciaMensual = transferencia;
        this.cantidadDireccionesCorreo = cantidadCorreos;

    }
```

Y se crea el *builder*:

```java
package com.guisho.software.patrones.builder;

import java.math.BigDecimal;

/**
 *
 * @author guisho.com, luishernan@gmail.com
 */

public class PaqueteDeHostingBuilder {

    private  PaqueteDeHosting paquete;

    public PaqueteDeHostingBuilder(String nombre, BigDecimal precio, int cantidadA

        this.paquete.setNombre(nombre);
```

```
        this.paquete.setPrecioAnual(precio);
        this.paquete.setCapacidadDeAlmacenamiento(cantidadAlmacenamiento);
        this.paquete.setTransferenciaMensual(transferenciaMesual);
        this.paquete.setCantidadDireccionesCorreo(cantidadCorreo);

    }

    public PaqueteDeHostingBuilder catidadSitiosPermitidos (int cantidad){

        this.paquete.setCantidadSitiosPermitidos(cantidad);
        return this;

    }

    public PaqueteDeHostingBuilder cantidadBaseDeDatos (int cantidad){

        this.paquete.setCantidadBaseDeDatos(cantidad);
       return this;

    }

    public PaqueteDeHostingBuilder accessoSsh(boolean acceso){

        this.paquete.setAccesoSsh(acceso);
        return this;

    }

    public PaqueteDeHostingBuilder panelControl (boolean panel){

        this.paquete.setPanelDeControl(panel);
        return this;

    }

    public PaqueteDeHostingBuilder codigoOferta(String codigo){

        this.paquete.setCodigoOferta(codigo);
        return this;

    }

    public PaqueteDeHostingBuilder ipPublica (String ip){

        this.paquete.setIpPublica(ip);
```

```
62          return this;
63
64      }
65
66  }
```

Si se observa el *builder* simplemente envuelve al objeto que creará, con una especie de métodos de acceso (parecido a un *JavaBean*) pero con la peculiaridad que se devuelve a sí mismo siempre. ¿En qué ayuda esto? Véase el cliente como crea un PaqueteDeHosting ahora:

```java
1   public static void main(String[] args) {
2
3       PaqueteDeHosting personal = new PaqueteDeHostingBuilder("personal",new Big
4
5       PaqueteDeHosting bronce =
6
7       new PaqueteDeHostingBuilder("bronce",new BigDecimal(200),100,1000,10).acce
8
9       PaqueteDeHosting plata =
10
11      new PaqueteDeHostingBuilder("plata",new BigDecimal(300),100,1000,50).acces
12
13      PaqueteDeHosting oro =
14
15      new PaqueteDeHostingBuilder("oro",new BigDecimal(500),100,4000,100).access
16
17  }
```

Esto es mucho más sencillo de leer (aparte que la línea se alarga un poco, aunque pueden hacecer varias líneas), y deja al objeto siempre en un estado consistente. Este método de construcción por medio de llamadas encadenadas se llama interfaces fluidas, y es el punto de inicio para muchos lenguajes como *Groovy*, que crean construcciones bastantes complejas a partir de *builders* sencillos que permiten muchas configuraciones.

De nuevo, este no es el Builder de GoF, pero es otro concepto de *builder*, que se expone para que se vea la flexibilidad que nuevos patrones pueden traer.

## 3.2 Lazy initialization

## 3.3 Factory

**TODO** definir mejor la teoría de este patrón.

Para ilustrar este patrón, se hará uso de un ejemplo. Supóngase que la tarea que se está por hacer es crear un traductor que devuelva los números del cero al diez en tres idiomas: inglés, español, y alemán. Existen muchísimas maneras de hacer esto. Al final, se desea un método que reciba un entero entre 0 y 10 y que devuelva una cadena con el nombre de dicho número en el idioma que se esté trabajando.

Una manera de resolver el problema puede ser algo así:

Java

```
1
2  public class MainClient {
3
4  public MainClient(){
5
6       }
7
8      public traducirNumero(String idioma, int numero){
9          if (idioma.equals("español")){
10           switch (numero){
11          case 0: return "uno";
12              case 1: return "dos";
13             ....
14            }
15         }
16
17         if (idioma.equals("english")){
18             switch (numero){
19              case 0: return "one";
20              case 1: return "two";
21             ....
22
23             }
24
25         }
26
27         if (idioma.equals("deutsch")){
28
29             switch (numero){
30              case 0: return "eins";
31              case 1: return "zwei";
32             ....
```

```
            }
        }

    }//traducirNumero


public static void main(String args[]){

    MainClient mc = new MainClient();
    System.out.println(mc.traducirNumero("espanol",1));

}

}//de la clase
```

El resultado del código anterior es: `uno`

Esta solución parece funcionar, y de hecho lo hace. Pero imagínese que ahora nos dicen sea desea la traducción de todos los números. El código comenzará a crecer desmedidamente, ya que ahora se hace necesario agregar código de lógica para cada idioma para escribir números como 752, 1233, etc.

Siguiendo la escuela de programación orientada a objetos, una solución natural puede ser el uso de herencia. Se define una clase abstracta Traductor, y para cada idioma se escribe una subclase de Traductor.

Java

```

public abstract class Traductor{
   public abstract String traducirNumero(int numero);

}
```

Ahora comienza la magia a aparecer: se va a crear una clase especializada para diccionario, que se encargará de traducir los números. Se tendrá una clase especializada para traducir los números al español, que iría algo así:

```java

public class TraductorEspanol extends Traductor {
    public TraductorEspanol(){
       super();
       ...
    }

    public String traducirNumero(int numero){

       switch(numero){
           case 1: return "uno";
           case 2: return "dos";
           ...

       }

    }

}
```

La clase para el inglés sería:

```java
public class TraductorIngles extends Traductor {

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

    }

    public String traducirNumero(int numero){

       switch(numero){
           case 1: return "one";
           case 2: return "two";
           ...

       }
    }
}
```

La del alemán sería similar. Ahora, en el momento de querer utilizar un diccionario, se llamaría de la siguiente

manera:

```java

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

Entonces, la clase MainClient cambiaría un poco y quedaría así:

```java
public class MainClient {
    public String  traducirNumero(int numero){
        Traductor traductor = null;
        if (idioma.equals("español")){
            traductor = new TraductorEspanol();
        }

        if (idioma.equals("ingles")){
            traductor = new TraductorIngles();
        }

        if (idioma.equals("aleman")){
            traductor = new TraductorAleman();
        }

        String toReturn traductor.traducirNumero(numero);
       return toReturn;

    }//traducirNumero

     public static void main(String args[]){
         MainClient mc = new MainClient();
         System.out.println(mc.traducirNumero("espanol",1));

     }

}//de la clase
```

Con este nuevo enfoque, ¿qué se ha ganado? Primero, el código es mucho más legible. Segundo es bastante más escalable: se puede agregar el traductor para el francés muy fácilmente. Tercero, se ha escondido la manera en la que se traduce a Tradúceme. Por ejemplo, puede ser que las traducciones a chino se vayan a traer a un *Web Service*. En ese caso TraductorChino se encargaría de hacer todo el ajetreo de conectarse a Internet y buscar el *web services*, pero los demás ni se enteran.

El *Factory Pattern* no ha aparecido, Es tiempo de irlo a llamar. Traduceme está haciendo algo que no le compete: está eligiendo la instancia de Traductor que quiere usar. Piénses que se usa el traductor en 100 lugares, entonces en cien lugares se tiene que buscar qué clase de Traductor vamos a instanciar. El patrón de fábrica (*factory pattern*) esconde esa lógica. Se va a agregar ahora la fábrica de traductores.

Java

```
1
2  public class TraductorFactory {
3
4       public TraductorFactory(){
5       }
6
7      public static Traductor createTraductor(int numero){
8
9          Traductor traductor = null;
10
11         if (idioma=="español"){
12             traductor = new TraductorEspanol();
13         }
14         if (idioma=="english"){
15             traductor = new TraductorIngles();
16         }
17         if (idioma=="deutsch"){
18             traductor = new TraductorAleman();
19         }
20         return traductor;
21
22     }
23
24 }//de la clase
```

¿Qué hace TraductorFactory? Simplemente elige, en base a los argumentos dados – en este caso el idioma- qué clase de traductor se instanciará. Traduceme de nuevo cambia y quedaría así:

Java

```
1  public class MainClient {
2
3       String idioma;
4       public static void main(Strin []args){
5        Traductor traductor = TraductorFactory.createTraductor("espanol");
6        System.out.println( traductor.traducirNumero(1) );
7      }//main
8
9  }//de la clase
```

MainClient se ha visto dramáticamente reducido, y su código es muy fácil de leer. Quien quiera usar un traductor simplemente hará llamar a `Traduceme` . `Traduceme` sabe el idioma que eligieron, pero no sabe que subclase de Traductor instanciar, pero sabiendo el idioma `TraductorFactory` sabe exactamente qué instancia de Traductor crear. Si la aplicación desea cambiar de idioma simplemente le envía otro parámetro a `Traduceme` y listo. También agregar idiomas es más manejable que antes.

El *Factory Pattern* esconde al usuario final del código la desición de qué sublclase instanciar, y promueve el encapsulamiento de las partes más variables del sistema. En términos generales, una fábrica abstracta consiste de las siguientes partes:

Un cliente, que es el que llama a la fábrica (en nuestro caso MainClient).

Una fábrica, que decide la clase a instanciar ( `TraductorFactory` ).

Un producto, lo que la fábrica devuelve (para nosotros las instancias de Traductor).

## 3.4 Abstract Factory

En la sección anterior se abordó el *Factory Pattern* (o *Factory Method Pattern*) . Ahora se tratará del *Abstract Factory Pattern*, que va un paso más allá: el *Abstract Factory* agrupa varios *Factory Methods*.

Básicamente lo que hace este patrón es unir varios Factory Methods, delegando la responsabilidad total sobre qué instancias crear a partir de datos comunes. En la sección anterior se utilizó el ejemplo de un pequeño traductor, que al recibir un número desplegaba su valor en español, inglés o alemán. Para mostrar el concepto del Abstract Factory se extenderá el mismo ejemplo.

Ahora se hará un sencillo reloj que muestra la hora actual. La hora puede ser desplegada en formato de 24 horas o puede ser desplegada en formato AM/PM. Recordando que es a manera de ejemplo, se utilizará la clase Date de una manera no aconsejable. Esto para omitir código de plomería extra, y facilidad de lectura. Como en el caso del diccionario, se hará una clase abstracta de Reloj y dos implementaciones para cada una de los formatos, y una clase que contenga el método del Factory Method. El código sería así:

Java

```
La clase Reloj:

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

La clase que da la hora en formato AM/PM:

```java
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();
        String tr;
        if (hora<=12){
            tr="Son las "+hora+":"+minutos+":"+segundos+" AM";
        } else {
            tr="Son las "+(hora-12)+":"+minutos+":"+segundos+" PM";
        }
        return tr;
    }

}
```

La que da la hora en formato de 24 horas:

```java

public class Reloj24Hrs extends Reloj {

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

        tr = "Son la(s) " + hora + ":" + minutos + ":" + segundos + " ";
        return tr;

    }

}
```

Ahora la clase que contiene la el método que elige las instancias. A diferencia de la sección anterior, ahora el

parámetro que recibe el método es un entero, que acepta los enteros especificados como constantes estáticas en la clase. Esto para no estar adivinando los parámetros que acepta el método:

```java

public class RelojFactory {

    public static final int RELOJ_AM_PM=0;
    public static final int RELOJ_24_HRS=1;

    public RelojFactory(){

    }

    public static Reloj createReloj(int tipoDeReloj){
        if (tipoDeReloj==RelojFactory.RELOJ_24_HRS){
            return new Reloj24Hrs();
        }

        if (tipoDeReloj==RelojFactory.RELOJ_AM_PM){
            return new RelojAmPm();
        }
        return null;
    }
}
```

Y finalmente la clase cliente, que será la usuario final:

```java

public class MainClient {

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

Hasta aquí se teinen dos fábricas: una de palabras, y la que recién creada que da la hora. Supóngase se solicita hacer un programa que despliegue la hora y los números de la manera en la que se expresan en cada país (una implementación súper elemental de *Locale* de Java). En Estados Unidos se despliegan los números en inglés, y la hora en formato AM/PM; mientras que en Guatemala se dicen los números en español y la hora en formato de 24 Horas.

Ahora se crea una *Abstract Factory*, que será llamada *Locale*.

```java
public abstract class AbstractLocaleFactory {

    public static final String US="ESTADOS_UNIDOS";
    public static final String GT="GUATEMALA"
    String pais;
    public abstract Traductor createTraductor();
    public abstract Reloj createReloj();

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

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

}
```

Esta fabrica tiene un par de métodos que devuelven un Reloj y un Traductor. Nótese que Reloj y Traductor son a su vez clases abstractas.

Ahora se procede a implementar la clase LocaleGuatemalaFactory, que se ve así:

```java

public class LocaleGuatemalaFactory extends AbstractLocaleFactory{

    public LocaleGuatemalaFactory(){
        this.pais=this.GT;
    }

    public Traductor createTraductor() {
        return TraductorFactory.createTraductor("espanol");
    }

    public Reloj createReloj() {
        return RelojFactory.createReloj(RelojFactory.RELOJ_24_HRS);
    }

}
```

Y la respectiva para Estados Unidos:

```java
public class LocaleEstadosUnidosFactory extends AbstractLocaleFactory{

    public LocaleEstadosUnidosFactory(){
        this.pais=AbstractLocaleFactory.US;
    }

    public Traductor createTraductor() {
        return TraductorFactory.createTraductor("ingles");
    }

    public Reloj createReloj() {
        return RelojFactory.createReloj(RelojFactory.RELOJ_AM_PM);
    }

}
```

Ahora en el cliente, si se desea las cosas como se verían en Guatemala, simplemente se hace lo siguiente:

```java
public class MainClient {

    public static void main(String[] args) {
        Reloj reloj = null;
        Traductor traductor = null;
        AbstractLocaleFactory localeFactory = new LocaleGuatemalaFactory();
        reloj = localeFactory.createReloj();
        traductor = localeFactory.createTraductor();
        System.out.println("--------");
        System.out.println("1="+traductor.traducirNumero(1));
        System.out.println(reloj.dameLaHora());

    }

}
```

El resultado de correr el codigo anterior es:

```
1=uno
Son las 21:50:17
```

Si se cambia la siguiente línea:

```java
        AbstractLocaleFactory localeFactory = new LocaleGuatemalaFactory();
```

Por esta:

```java
        AbstractLocaleFactory localeFactory = new LocaleEstadosUnidosFactory();
```

Se tiene como resultado:

```
1=one
9:52:56 PM
```

Este es un ejemplo sencillo. Pero imagínese que se desea implementar hacer un *Locale* para cada país, y en el tener más cosas como: la nomenclatura de moneda, el sistema de numeración, el manejo de fechas, kilómetros o millas, etc. Con el *Abstract Factory Pattern* es muy sencillo agregar cada nuevo país, o cada nueva característica del Locale. Pero sobre todo el código es MUY legible y FACILMENTE extensible. Alguien que nunca ha visto estas piezas de código puede entender como hacer un nuevo país.

Entonces, el *Abstract Factory Pattern* ayuda mucho en casos donde se han de manejar familias de objetos. Al inicio no siempre es obvia su implementación, pero siempre está el recurso de del refactoring, en el cual salen nuevas maneras sencillas de hacer las cosas. Este, como muchos patrones, requieren escribir un poco más de código al principio, pero reduce el esfuerzo a largo plazo porque hay menos código repetido.

## 3.5 Prototype

Como los demás patrones creacionales, este patrón sustituirá para el cliente la palabra clave *new* por otra forma de crear objetos. En este caso específico la creación se hará sobre objetos que son complicados de crear, que para evitar usar *new* serán clonados a partir de una instancia ya existente. Este patrón intenta, como su nombre lo indica, clonar el ADN de un objeto a otra. Es decir cada instancia del objeto se obtendrá a partir de un prototipo.

En el caso específico de Java ya se tiene mucho camino ganado, porque Java provee la interfaz *Clonable* con el propósito de crear clones en mente. Se verá también una implementación sin usar esta interfaz para comprender completamente la idea detrás del prototipo.

Véase un ejemplo utilizando *Clonable*

```java
public class Copiame implements Cloneable {

          Object clone = null;

  public Object clone() {
          try {
              clone = super.clone();
          } catch (CloneNotSupportedException e) {
              e.printStackTrace();
          }

          return clone;

}
```

`Clonable` es una interfaz vacía, que requiere se implemente `clone`, que hace una copia de los valores de los campos de un objeto, no de los objetos a los que apunta. En otras palabras, el objeto clonado apuntará a los mismos objetos que el objeto antiguo apuntaba. Otra cosa a tomar en cuenta es que `clone()` devuelve siempre un `Object`. También nótese que `clone()` es un método protected que puede ser llamado solo por la misma clase o el paquete que la contiene.

Dicho esto, ¿para qué sirve clonar objetos? Sirve para copiar objetos que tardan mucho en crearse, o que son complejos de crear. Por ejemplo una lista grande que es costosa en tiempo obtener y que se desea ordenar de dos maneras distintas.

Véase este ejemplo, en el que se procederá hacer una clase `Persona` y luego llenar la persona con los datos de dos hermanos: Juan y María que serán ingresados a un sistema x.

```java

package com.guisho.software.patrones.prototype;

/**

 *

 * @author guisho.com, luishernan@gmail.com

 */

public class Persona implements Cloneable{

    public String nombres;
    public String appellidos;
    public String nombrePadre;
    public String nombreMadre;
    public String direccion;
    public String telCasa;
    public String nacionalidad;
    public String ciudadEnQueVive;
    public String nombreMascota;
    public Persona(){

    }

/*Setters y geters....*/

}
```

Y un cliente que crea la `Persona`:

```java
    public static void main(String[] args) {

        Persona juan = new Persona();
        juan.setNombres("Juan José");
        juan.setAppellidos("Pérez Ramirez");
        juan.setNombrePadre("Juan Pérez Martinez");
        juan.setNombreMadre("María Ramirez");
        juan.setDireccion("9a. Ave. 43-12 Z.1");
        juan.setTelCasa("34567890");
        juan.setNacionalidad("Guatemalteca");
        juan.setCiudadEnQueVive("Guatemala");
        juan.setNombreMascota("Pepito");

        //Hacer algo con Juan, ingresarlo al sistema, etc...
        System.out.println("Ingresando al sistema :"+juan.toString());

        Persona maria = (Persona)juan.clone();
        maria.setNombres("María Inés");

        System.out.println("Ingresando al sistema :"+maria.toString());
        //ingresar a Maria al sistema....

    }
```

Como puede notarse, no fue necesario ingresar todos los campos de María sino, sólo aquellos que la diferenciaban de su hermano Juan. Hay que recordar que si hubiesen objetos en el ejemplo estos se clonan por referencia, es decir las referencias de ambos objetos son las mismas.

Ahora se implementará una clonación desde cero (sin utilizar la interfaz `Clonable`). Para ello se añade a `Persona` el siguiente método:

```java

    public Persona creaPrototipo(){

        Persona p = new Persona();
        p.setNombres(this.nombres);
        p.setAppellidos(this.appellidos);
        p.setNombrePadre(this.nombrePadre);
        p.setNombreMadre(this.nombreMadre);
        p.setDireccion(this.direccion);
        p.setTelCasa(this.telCasa);
        p.setNacionalidad(this.nacionalidad);
        p.setCiudadEnQueVive(this.ciudadEnQueVive);
        p.setNombreMascota(this.nombreMascota);
        System.out.println("Creado prototipo de Persona");
        return p;

    }
```

Esto permite que el cliente pueda hacer la siguiente llamada:

```java
        Persona maria = juan.creaPrototipo();
```

La idea básica del patrón es sencilla: crear una copia de un objeto para ahorrarse los pasos de su creación, o para optimizar accesos o procesos que ya se hicieron en un objeto similar y crear una copia del objeto ya con esos datos ingresados.

## 3.6 Singleton

El *Singleton Pattern*, a diferencia de los otros patrones de diseño creacionales, no se encarga de la creación de objetos en sí, sino que se enfoca en la restricción en la creación de un objeto. Este patrón es ampliamente utilizado por muchos *frameworks*, y también es uno de los más fáciles de aprender y utilizar.

Siempre que se crea un objeto nuevo (en Java con la palabra reservada `new`) se invoca al constructor del objeto para que cree una instancia. Por lo general los constructores son públicos. El *singleton* lo que hace es convertir al constructor en privado, de manera que nadie lo pueda instanciar. Entonces, si el constructor es privado, ¿cómo se instancia el objeto? Pues a través de un método público y estático de la clase. En este método se revisa si el el objeto ha sido instanciado previamente. Si no ha sido instanciado, llama al constructor y guarda el objeto creado en una variable estática del objeto. Si el objeto ya fue instanciado anteriormente, lo que hace este método es devolver la referencia al estado creado anteriormente.

En los patrones anteriores se utilizó un `Traductor`. Imagínese que el traductor carga a memoria no sólo números, pero también diez mil palabras obtenidas a través de un archivo de texto o un *web service*. Cada vez que este objeto se cree utilizará mucho espacio en memoria. Además, si se usa un *web service* para cargarlo, cada carga consume muchos recursos de red y tarda mucho en terminarse de construir.

`Traductor` estará disponible para toda la aplicación, y en cualquier lado que se despliegue un texto será invocado. No tendría mucho sentido construir un `Traductor` cada vez que se lo quiera utilizar. Lo más sano sería utilizar un sólo `Traductor` para toda la aplicación. ¿Cómo lograrlo? A través de un Singleton. Omitiendo la lógica del objeto, el código que se debería usar quedaría algo así:

Java

```
public class TraductorPesado {

    private static boolean instanciated = false;

    private static TraductorPesado traductorInstance;

    /**
     * Notar que el constructor es privado!
     */
    private TraductorPesado() {
        //cargar un diccionario a memoria  a través de un WebService.
    }

    public static TraductorPesado getTraductor() {
        if (!TraductorPesado.instanciated) {
            TraductorPesado.traductorInstance = new TraductorPesado();
            TraductorPesado.instanciated = true;
        }
        return TraductorPesado.traductorInstance;
    }

    public String translate(String toTranslate) {

        //mucho código bonito va aquí
        return null;
    }

}```

En cualquier lugar de la aplicación que se quiera utilizar hacer una traducción se

```java
TraductorPesado.getTraductorPesado().translate("unaPalabra");
```

Con esto se logra que nadie, dentro del ambiente de la máquina virtual, pueda hacer lo siguiente.
`java TraductorPesado t = new TraductorPesado();`

Es un gran beneficio porque se puede controlar mejor la manera en la que `TraductorPesado` puede ser usada. Evita malos usos de la clase y se asegura que a lo más hay una instancia del objeto en toda la aplicación.
```

Hay muchas maneras de crear los Singletons, que pueden complicarse. En este ejemplo se utiliza un booleano estático, pero no siempre es necesario, se pudo haber inicializado `traductorInstance` como `null`, y en vez de verificar la variable booleana, verificar si la instancia es null o no.

TODO Do Code ```java public class Traductor{

```
  private static  Traductor traductorInstance=null;

  /**

   *Notar que el constructor es privado!

  */

  private Traductor(){

      //cargar un diccionario a memoria  a través de un WebService.

 }

public static Traductor getTraductor(){

       if (! Traductor.INSTANCE==null){

             Traductor.traductorInstance= new Traductor();

       }

      return Traductor.traductorInstance;

}

public String translate(String toTranslate){

   //mucho código bonito va aquí

}
```

}```

O, para hacer las cosas más sencillas (que no siempre conviene) se podría haber decidido evitar la evaluación en `getTraductorPesado` y crear el objeto cuando al momento de declararlo:

```java

public class Traductor{

      private static  Traductor traductorInstance=new Traductor();

      /**
       *Notar que el constructor es privado!
      */
      private Traductor(){
          //cargar un diccionario a memoria  a través de un WebService.
     }

    public static Traductor getTraductor(){
           return Traductor.traductorInstance;
    }

    public String translate(String toTranslate){
       //mucho código bonito va aquí
    }

}
```

Fácil ¿no? Mmm, pues se puede complicar. En Java por ejemplo, todavía se podría obtener una copia de traductor así:

```java
Traductor t = (Traductor)Traductor.getTraductor().clone();

Para evitar esto tendríamos que añadir las siguietnes líneas a nuestra clase Tradu

     public Object clone() throws CloneNotSupportedException {

         throw new CloneNotSupportedException();

       }

```

También alguien podría extender la clase y volver público el constructor. Para evitar esto sería buena idea declarar nuestra clase como final.

Hay que tener especial cuidado cuando el Singleton se utiliza en un ambiente multi hilos, porque puede crear problemas si no se implementa de la manera adecuada. En *Java* es posible que tengamos que meter algún

synchronized por ahí para evitar problemas.

Concluyendo, la idea central del Singleton es esa: asegurar de que exista tan solo una instancia del objeto en toda la aplicación. Hay muchas maneras de implementar un Singleton (aquí solo vimos algunas). Es un patrón muy aplicado en Java, aunque, como todos los patrones, se puede implementar en cualquier lenguaje orientado a objetos. También se pueden hacer cosas interesantes uniendo el Singleton con otros patrones creacionales (recordemos que el singleton no busca crear, sino que limitar la creación).

# 4. Patrones Estructurales.

Los patrones estructurales (*structural patterns*) podrían llamarse patrones de relaciones, o algo parecido, porque su principal función es facilitar y mejorar las relaciones entre objetos.

En el mundo de objetos, la creación de instancias es muy importante, pero tan importante como la creación, es la manera en la que instancias de objetos se comunican entre sí. Un diseño estandarizado y bien pensado puede facilitar mucho las cosas durante el desarrollo de un sistema complejo. Es fácil que una solución compleja, especialmente si la complejidad crece en el tiempo, posea una construcción extraña. Aunque hasta cierta medida esto siempre pasará, el tener en mente patrones creacionales ayudará a reducir la complejidad, y la rareza de cada implementación. Una buena implementación de código aspira a ser lo más estándar posible, de manera que cualquier programador pueda entenderla en el futuro.

Debe tenerse cuidado, sin embargo, en el hacer ingeniería de más. Los patrones de diseño han de usarse ahí donde resuelvan problemas, nunca para hacer las cosas más complicadas. Por eso su correcto uso es más un arte que una ciencia. Los patrones son una herramienta de diseño muy poderosa, pero hay que saberla utilizar. El tiempo y la experiencia suelen ser los mejores acompañantes para un diseñador de software, porque sólo el ellos le darán el colmillo para utilizar sus herramientas adecuadamente.

Con mucha frecuencia el diseño de un dominio específico para un sistema inicia con nociones sumamente intuitivas. Conforme el diseño se va destilando y mejorando, se agregan soluciones específicas de software (como patrones), que lo pueden ir complicando (con el propósito de hacerlo más manejable). Generalmente los patrones estructurales entran en la jugada en etapas medias del diseño, y muchísimas veces son el resultado de refactorización (*refactoring*).

## 4.1 Decorator

En muchas ocasiones, una vez creado un objeto, la dificultad consiste en agregarle funcionalidad una vez creado. Esto se vuelve especialmente complicado cuando el objeto tiene una serie de combinaciones que es posible que lo "decoren". En el caso de la programación orientada a objetos, una manera natural de lidiar con la complejidad y las combinaciones, es utilizar la herencia. Sin embargo, como estamos a punto de ver, esto

puede llevar a problemas muy fuertes.

El decorador es un envolvente de la clase original. Es decir, el decorador y el decorado implementan la misma interfase. Esto permite al decorador agregar funcionalidad a cada método (declarado en la interfaz) de la clase decorada. Esto permite agregar funcionalidad a un objeto sin modificar su estructura.

Supóngase que se está vendiendo boletos de avión. Como es de esperar, al momento de hacer el modelo de dominio, en algún lado ha de aparecer una interfaz, o una clase abstracta llamada `BoletoDeAvion`. Para simplificar, esta interfaz (que para utilizar el patrón decorador es mejor utilizar una interfaz) tendrá únicamente dos métodos `getAmenidades()` y `getCosto()`. Estos dos métodos, como es de esperarse, devolverán las amenidades y los costos de un avión. La aerolínea para la que se está haciendo el programa, tiene diferentes tipos de boletos. Por ejemplo tiene clase ejecutiva, tiene primera clase, y tiene clase económica. En la clase económica existen varias amenidades: existe la posibilidad de utilizar TV Satelital, de utilizar WiFi, de elegir una silla con más espacio, y ahora comenzó a ofrecerse el ingreso preferencial.

Utilizando de manera correcta, el equipo de desarrollo crea las siguientes clases que implementan `BoletoDeAvion`: `BoletoPrimeraClase`, `BoletoClaseEjecutiva` y `BoletoClaseEconómica`. Ahora bien, en la clase económica ninguna de las amenidades anteriormente mencionadas se ofrece de manera estándar, sino que se ofrecen con un costo extra. Uno de los programadores, siguiendo las buenas prácticas comienza entones a desarrollar las siguientes clases `BoletoClaseEcononicaWifi`, `BoletoClaseEconomicaTVSatelital`, `BoletoClaseEconomicaSillaGrande`, `BoletoClaseEconomicaIngresoPreferencial`, que extienden a `BoletoDeAvion`. La complejidad comienza cuando las combinaciones comienzan a aparecer, ya que también se pueden hacer combinaciones de amenidades: se puede tener TV Satelital y silla más grande. O Ingreso Preferencial y WiFi. La cantidad de combinaciones empieza a crecer con cada amenidad que se agrega, lo que hace poco práctico un esquema de extender clases clásico. Es aquí donde entra a jugar un papel beneficioso el Patrón Decorador.

Lo primero que se hace, es se crea la interfaz:

```java
public interface BoletoDeAvion {

    String getAmenidades();
    Double getCosto();
}
```

Después se crea la clase que implementa la interfaz

```java
public class BoletoDeCabinaEconomica implements BoletoDeAvion {


    public BoletoDeCabinaEconomica() {
    }

    @Override
    public String getAmenidades() {
        return "Boleto Sencillo Cabina Económica";
    }

    @Override
    public Double getCosto() {
        return 1000.0;
    }

}
```

Como puede verse, esta clase implementa la funcionalidad básica de la clase concreta. Hasta este momento, una implementación tradicional orientada a objetos. Ahora es cuando comienza a aparecer el patrón decorador. Primero, se crea el decorador en sí, que es una clase abstracta que implementa la misma interfaz que el objeto a decorar, en este caso `BoletoDeAvion` :

```java
public abstract class BoletoDeAvionDecorator implements BoletoDeAvion {

    private BoletoDeAvion boleto;

    public BoletoDeAvionDecorator(BoletoDeAvion boleto) {
        this.boleto = boleto;
    }

    public abstract String getAmenidades();
    public abstract Double getCosto();

    public BoletoDeAvion getBoleto(){
        return this.boleto;
    }
}
```

Y se comienzan a crear los decoradores, en este caso se muestra el de Tv Satelital: ```java public class TvSatelital extends BoletoDeAvionDecorator {

```
public TvSatelital(BoletoDeAvion boleto) {
    super(boleto);
}

@Override
public String getAmenidades() {
    return this.getBoleto().getAmenidades() + ", Tv Satelital";
}

@Override
public Double getCosto() {
    return this.getBoleto().getCosto() + 75.0;
}
```

}```

Como extienden de una clase abstracta, es obligado implementar los métodos, y por lo tanto es ahí donde es posible agregar funcionalidad. Ahora véase otro decorador, el de IngresoPreferencial:

Java
```
public class IngresoPreferencial extends BoletoDeAvionDecorator {

    public IngresoPreferencial(BoletoDeAvion boleto) {
        super(boleto);
    }

    @Override
    public String getAmenidades() {
        return this.getBoleto().getAmenidades() + ", Ingreso Preferencial";
    }

    @Override
    public Double getCosto() {
        return this.getBoleto().getCosto() + 101.0;
    }
}
```

Como se puede percibir, el mismo concepto. Ahora mostramos la clase principal que llama a todos los objetos:

```java
        BoletoDeAvion boleto = new SillaGrande(
         new IngresoPreferencial(new BebidasAlcoholicas(new TvSatelital(new Boletc

        System.out.println("Hola, Bienvenido a Acatenango Airlines. ");
        System.out.println("Amenidades <"+boleto.getAmenidades()+"> ");
        System.out.println("Costo $<"+boleto.getCosto()+">");
```

En este caso se imprime lo siguiente:

```
Hola, Bienvenido a Acatenango Airlines.
Amenidades <Boleto Sencillo Cabina Económica, Tv Satelital, Bebidas Alcolholicas I
Costo $<1501.0>
```

Puede notarse el poder que da el decorador, ya que evita el crear una gran cantidad (potencialmente inmanejable) de implementaciones concretas, dejando esa responsabilidad al decorador, permitiendo al usuario final del código, realizar las combinaciones que desee, sin preocuparse de crear complejidad. De esto se tratan los patrones de diseño: facilitar las implementaciones.

## 4.2 Composite

## 4.3 Adapter

Se puede utilizar en muchos contextos y es de especialidad utilidad cuando se utilizan códigos o librerías ajenos sobre el que no se tiene control. Este patrón se le conoce como *adapter* (adaptador) , aunque algunos lo llaman también *wrapper* (envoltorio). Ambos nombres tienen bastante sentido y explican el por qué de este patrón.

Piénsese en el siguiente problema: en muchos países se utilizan las espigas redondas en los toma corrientes, y en otros las espigas planitas. En ocasiones se tiene un dispositivo que tiene 3 espigas y el toma corrientes 2. Sin embargo el aparato que se desea conectar "entiende" la corriente eléctrica, “la acepta”, aunque la interfaz para conectarse a ella sea una distinta a la que posee. ¿Qué se hace en estos casos? Se usa un adaptador, o un convertidor, o un transformador, cómo le desee llamar. Lo mismo sucede con el software. A menudo un equipo de desarrollo se encuentra con librerías que pueden ser de utilidad, pero que existe la necesidad de adaptarse a ellas. En esos casos existen dos opciones: modificar todo el código propio para que se adapte a la librería, o se puede crear un adaptador que traduzca lo propio a lo de ellos y lo de ellos a lo propio. Un ejemplo sencillo en el mundo Java es el de los `Enumeration` y los `Iterators` . Ambos tienen un

`hasNext()` o un `hasMoreElements()` que hacen lo mismo; al igual que un `next()` y un `nextElement()` que hacen lo mismo. Imagínese que todo se desee manejar con `Iterators`, se puede crear un adaptador que “convierta” entre `Iterator` y `Enumeration`. Esto no es recomendable, pero ilustra bien el uso de un adaptador.

Imagínese se tiene un sistema que maneja autos, barcos, aviones y parecidos. Generalmente los motores que se usan son de gasolina, pero las nuevas tendencias han popularizado los motores eléctricos. Se tiene que el proveedor de vehículos eléctricos provee sus librerías para el motor eléctrico que es prácticamente igual a las implementaciones propias pero con otros nombres (prender en vez de encender, "mover más rápido" en vez de acelerar, etc.), y tiene una restricción extra: para poder acelerar o detener el motor, este tiene que estar conectado. Y surge el problema ¿cómo haver para que las librerías propias puedan hacer uso del motor eléctrico? Una solución es re-escribirlas todas, pero el tiempo y costo de hacer eso es muy alto.

Entonces, se tiene:

Java

```
1
2   package com.guisho.software.patrones.adapter;
3
4   public abstract class Motor {
5
6      abstract public void encender();
7
8      abstract public void acelerar();
9
10     abstract public void apagar();
11
12     //...mas metodos que hacen muchas cosas
13
14  }
15  ```
16
17  Y se tienen las implementaciones de algunos motores, por ejemplo el motor económi
18
19  ```java
20
21  package com.guisho.software.patrones.adapter;
22
23  public class MotorEconomico extends Motor {
24
25      public MotorEconomico(){
26
27          super();
28
```

```
29          System.out.println("Craendo motor economico");
30
31      }
32
33      public void encender() {
34
35          System.out.println("Encendiendo motor economico.");
36
37      }
38
39      public void acelerar() {
40
41          System.out.println("Acelerando motor economico.");
42
43      }
44
45      public void apagar() {
46
47          System.out.println("Apagando motor economico.");
48
49      }
50
51  }
```

Y el motor gastón (que ya no se vende tanto!):

Java

```
1
2   package com.guisho.software.patrones.adapter;
3
4   public class MotorGaston extends Motor {
5
6       public MotorGaston(){
7
8           super();
9
10          System.out.println("Creando el motor gaston");
11
12      }
13
14      public void encender() {
15
16          System.out.println("Bum, bum....encendiendo motor gaston");
17
```

```
    }

    public void acelerar() {

        System.out.println("Buuuuuuuuuuuum, acelerando y gastando muuuucha gas");

    }

    public void apagar() {

        System.out.println("Apagando motor gaston");

    }

}
```

Estas clases siempre han funcionado bien, y de hecho tienen muchas cosas como ser

```java

        Motor motor = new MotorEconomico();

        motor.encender();

        motor.acelerar();

        motor.apagar();

       //hacer mas cosas....

        motor = new MotorGaston();

        motor.encender();

        motor.acelerar();

        motor.apagar();

```

Ahora la empresa que construye motores eléctricos envía su propia implementación

```java
public class MotorElectrico {
```

```
63
64      private boolean conectado = false;
65
66      public MotorElectrico() {
67
68          System.out.println("Creando motor electrico");
69
70          this.conectado = false;
71
72      }
73
74      public void conectar() {
75
76          System.out.println("Conectando motor eléctrico");
77
78          this.conectado = true;
79
80      }
81
82      public void activar() {
83
84          if (!this.conectado) {
85
86              System.out.println("No se puede activar porque no esta conectado el m
87
88          } else {
89
90              System.out.println("Esta conectado, activando motor electrico....");
91
92          }
93
94      }
95
96      public void moverMasRapido() {
97
98          if (!this.conectado) {
99
100             System.out.println("No se puede mover rapido el motor electrico porqu
101
102         } else {
103
104             System.out.println("Moviendo mas rapido...aumentando voltaje");
105
106         }
107
```

```
    }

    public void detener() {

        if (!this.conectado) {

            System.out.println("No se puede detener motor electrico porque no est

        } else {

            System.out.println("Deteniendo motor electrico");

        }

    }

    public void desconectar() {

        System.out.println("Desconectando motor electrico...");

        this.conectado = false;

    }

}
```

Como puede verse, este motor hace lo mismo que la implementación propia, pero de manera y con llamadas un poco diferentes. ¿Cómo se puede hacer para integrar este `MotorEléctrico` al resto del sistema? Con un adaptador o *adapter*.

El adaptador "envuelve" al objeto extraño (por eso le llaman *wrapper*).

El adaptador se escribiría así:

Java

```
package com.guisho.software.patrones.adapter;

public class MotorElectricoAdapter extends Motor{

    private MotorElectrico motorElectrico;

    public MotorElectricoAdapter(){

        super();
```

```
10
11          this.motorElectrico = new MotorElectrico();
12
13          System.out.println("Creando motor Electrico adapter");
14
15      }
16
17      public void encender() {
18
19          System.out.println("Encendiendo motorElectricoAdapter");
20
21          this.motorElectrico.conectar();
22
23          this.motorElectrico.activar();
24
25      }
26
27      public void acelerar() {
28
29          System.out.println("Acelerando motor electrico...");
30
31          this.motorElectrico.moverMasRapido();
32
33      }
34
35      public void apagar() {
36
37          System.out.println("Apagando motor electrico");
38
39          this.motorElectrico.detener();
40
41          this.motorElectrico.desconectar();
42
43      }
44
45  }
46
```

El adapter se encarga no sólo de corregir los nombres de los métodos, sino también de cosas como conectar y desconectar el motor, cosas que a la implementación propia no le importan. Pero lo más importante es que ahora se puede utilizar esta implementación de Motor en el sistema propioutilizando la implementación de ellos. Por ejemplo, se pueden hacer cosas como esta:

```java

        Motor motor = new MotorEconomico();
        motor.encender();
        motor.acelerar();
        motor.apagar();
        motor = new MotorGaston();
        motor.encender();
        motor.acelerar();
        motor.apagar();
        motor = new MotorElectricoAdapter() ;
        motor.encender();
        motor.acelerar();
        motor.apagar();
```

El patrón adaptador aparece en todos lados, aunque muchas veces no se le llama adapter específicamente.

TAJUMULCO

##4.4 Bridge

##4.5 Facade

##4.6 Proxy

##4.7 Module

# 5. Patrones de comportamiento.

## 5.1 Observer

## 5.2 Chain of responsibility

## 5.3 Command

## 5.4 Interpreter

## 5.5 Strategy

## 5.6 Template

## 5.7 Visitor

# 6. Conclusiones y recomendaciones

TODO Sobre el uso del lenguaje, ingles o espanol? Recordar que los patrones son sólo un tool. Recordar que los patrones aquí expuestos son para OOP, no para funcional u otros paradigmas. Afilar el machete