Patterns Creacionales

Por
Vanesa Smotrizky

con aportes de
Fernando Dodino

Versión 2.0
Mayo 2008

Índice

Creational Patterns

Hasta ahora, vimos como los mecanismos de herencia, composición, y delegación, nos ayudan a asegurar un menor acoplamiento entre clases, componiendo comportamientos en forma polimórfica para permitir que el diseño sea fácilmente mantenible y robusto ante cambios en los requerimientos.

Y los Patterns traen consigo estas ideas de diseño, solucionando problemas puntuales en un sistema.

Vimos que estos problemas podían ser:

- De Comportamiento (ej. Strategy, Observer)
- Estructurales (ej. Decorator)

Ahora vamos a analizar los problemas CREACIONALES, que son aquellos que surgen de la Instanciación en un sistema.

Problemas Creacionales

En un sistema simple, la instanciación de un objeto podría ser:
Poligono triangulo=new Triangulo();

De aquí surgen un par de consideraciones:

- La instanciación del Triangulo es fácil, ya que Triangulo no es un objeto complejo para construir.
- No hay mayores inconvenientes cuando la instanciación se referencia desde un solo lugar (algún método main de una clase Test).

Esto hace que no tengamos que preocuparnos por la clase instanciada.

Pero desde ya que, el hecho de incluir en el código del cliente el nombre de la clase concreta que se crea (Triángulo), produce acoplamiento, ya que si cambia el nombre de la misma y la instanciación se produce en muchos puntos del diseño, habrá que refactorizar, cambiar el nombre de esa clase en todos esos lugares.

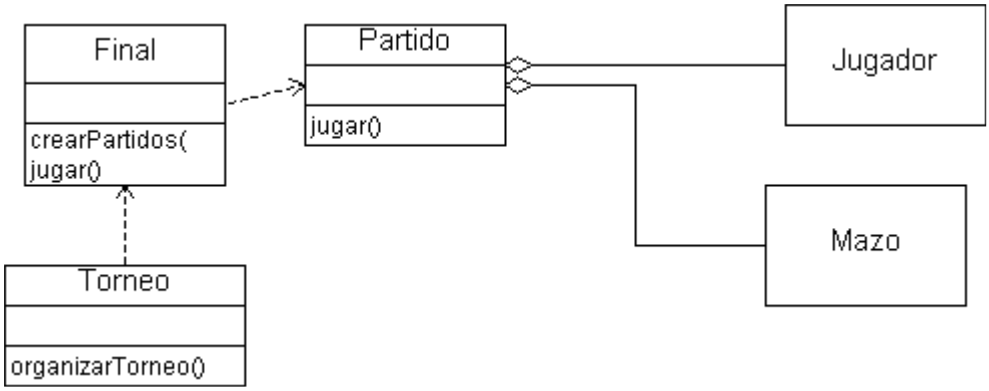
Cuando empezamos a diseñar sistemas más complejos, pueden surgir situaciones como éstas:

- El objeto a crear requiere de la composición de otros, y la forma de crearlo no es tan trivial.
- Todos los objetos creados en el sistema deben ser uniformes, o sea de un mismo tipo o familia.

Dicho esto, vamos a empezar a analizar un ejemplo en el cual se van a ver reflejados estos problemas y la solución brindada por cada pattern.

Ejemplo Torneo de Cartas

Un torneo de juegos de cartas podría modelarse de la siguiente manera:



Torneo tiene la responsabilidad de ir armando las distintas etapas del mismo (por cuestiones de tiempo vamos a ocuparnos sólo de la Final)

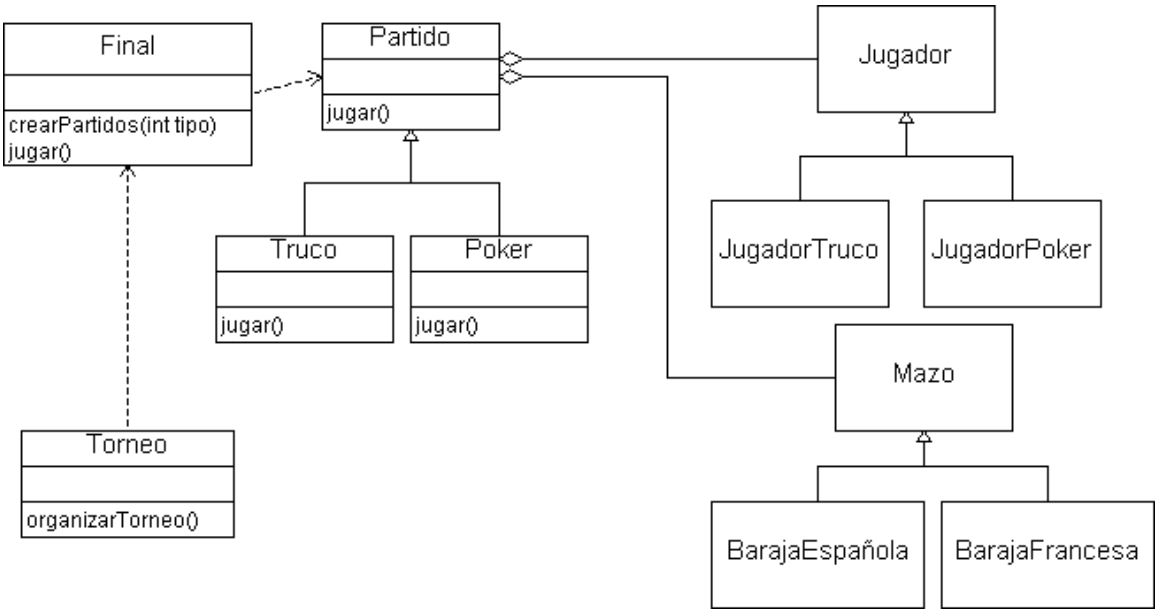
Final tiene la responsabilidad de crear dos partidos
Torneo va a crear una Final.
Partido es un objeto compuesto por 2 jugadores y un Mazo.
Torneo crea una final para luego llamar al método jugar().

```
public class Torneo {

   public Torneo(){
   }

   public void organizarTorneo(){

      Final finalTorneo=new Final();
      finalTorneo.crearPartidos();
      finalTorneo.jugar();

   }
}

public class Final {

   private Collection partidos;

   public Final() {
      partidos=new ArrayList();
   }

   public void crearPartidos(){
      Jugador jugadorA=new Jugador();
      Jugador jugadorB=new Jugador();
      Mazo mazo1=new Mazo();
      Partido partido1=new Partido(jugadorA,jugadorB,mazo1);
      partidos.add(partido1);
```

```
        Jugador jugadorC=new Jugador();
        Jugador jugadorD=new Jugador();
        Mazo mazo2=new Mazo();
        Partido partido2=new Partido(jugadorC,jugadorD,mazo2);
        partidos.add(partido2);
    }


    public void jugar(){
        Iterator itPartidos = partidos.iterator();
        while (itPartidos.hasNext()) {
            Partido partido = (Partido)itPartidos.next();
            partido.jugar();
        }
    }
}
```

Evidentemente, el método crearPartidos() está muy ligado el tipo de instancias que crea, ya que referencia directamente el nombre de las clases concretas de Jugador, Mazo y Partido.

Veamos que ocurre si se desean instanciar Partidos más específicos: Truco o Poker.



Al tener que crear Partidos de Truco y Poker, debemos crear para cada uno, su correspondiente Jugador y Mazo.

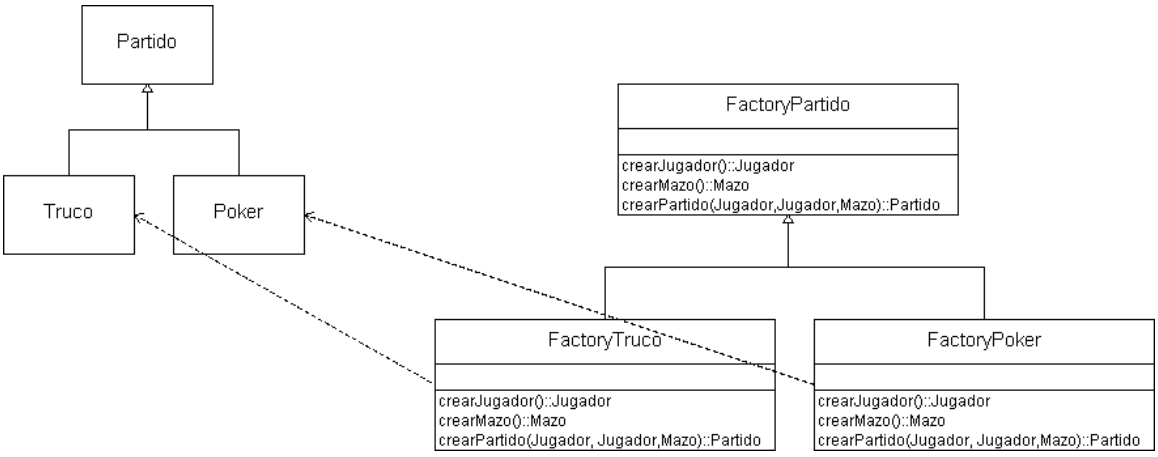
```
public void crearPartidos(int tipo){

   if(tipo==DEFAULT){
      Jugador jugadorA=new Jugador();
      Jugador jugadorB=new Jugador();
      Mazo mazo1=new Mazo();
      Partido partido1=new Partido(jugadorA,jugadorB,mazo1);
      partidos.add(partido1);

      Jugador jugadorC=new Jugador();
      Jugador jugadorD=new Jugador();
      Mazo mazo2=new Mazo();
      Partido partido2=new Partido(jugadorC,jugadorD,mazo2);
      partidos.add(partido2);
   }

   if(tipo==TRUCO){
      Jugador jugadorA=new JugadorTruco();
      Jugador jugadorB=new JugadorTruco();
      Mazo mazo1=new BarajaEspañola();
      Partido partido1=new Truco(jugadorA,jugadorB,mazo1);
      partidos.add(partido1);

      Jugador jugadorC=new JugadorTruco();
      Jugador jugadorD=new JugadorTruco();
      Mazo mazo2=new BarajaEspañola();
      Partido partido2=new Truco(jugadorC,jugadorD,mazo2);
      partidos.add(partido2);
    }

   if(tipo==POKER){
      Jugador jugadorA=new JugadorPoker();
      Jugador jugadorB=new JugadorPoker();
      Mazo mazo1=new BarajaFrancesa();
      Partido partido1=new Poker(jugadorA,jugadorB,mazo1);
      partidos.add(partido1);

      Jugador jugadorC=new JugadorPoker();
      Jugador jugadorD=new JugadorPoker();
      Mazo mazo2=new BarajaFrancesa();
      Partido partido2=new Poker(jugadorC,jugadorD,mazo2);
      partidos.add(partido2);
   }
}
```

Al tener que instanciar una serie de objetos de una misma especie, no podemos reutilizar el método anteriormente definido, hay que verificar que especie de partido es para saber que conjunto de objetos voy a instanciar.

¿Cuál es el problema? Estamos referenciando directamente el nombre de la clase concreta, y eso hace que estemos ligados a ella, y por lo tanto, acoplados.

El Pattern:

ABSTRACT FACTORY nos da la siguiente solución:




```
    public void crearPartidos(FactoryPartido factory){

      Jugador jugadorA=factory.crearJugador();
      Jugador jugadorB=factory.crearJugador();
      Mazo mazo1=factory.crearMazo();
      Partido partido1=factory.crearPartido(jugadorA,jugadorB,mazo1);
      partidos.add(partido1);

      Jugador jugadorC=factory.crearJugador();
      Jugador jugadorD=factory.crearJugador();
      Mazo mazo2=factory.crearMazo();
      Partido partido2=factory.crearPartido(jugadorC,jugadorD,mazo2);
      partidos.add(partido2);
   }
```

Fíjense como en todos los lugares donde teníamos la instancia concreta de un objeto, ahora tenemos un llamado a un objeto Factory, que nos devuelve un objeto de tipo Partido, o sea que en ningún momento sabemos con qué instancia concreta estamos trabajando. Solo conocemos su clase abstracta.

Esto nos permitió eliminar todo el código repetido, ya que llamamos polimórficamente a un factory, del cual no sabemos su clase concreta, podría ser FactoryPartido, FactoryTruco o FactoryPoker.

En el código del cliente:

```
      FactoryPartido factory=new FactoryTruco();
      Final finalTorneo=new Final();
      finalTorneo.crearPartidos(factory);
      finalTorneo.jugar();
```

```
public class FactoryPartido {

   public FactoryPartido() {
   }

   public Jugador crearJugador(){
```

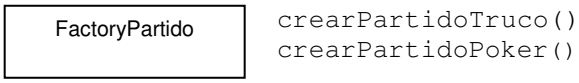
```
        return new Jugador();
    }

    public Mazo crearMazo(){
        return new Mazo();
    }

    public Partido crearPartido(Jugador jugadorA,Jugador jugadorB, Mazo
mazo){
        return new Partido(jugadorA,jugadorB,mazo);
    }

}


public class FactoryPoker extends FactoryPartido {

    public FactoryPoker() {
        super();
    }

    public Jugador crearJugador() {
        return new JugadorPoker();
    }

    public Mazo crearMazo() {
        return new BarajaFrancesa();

    }

    public Partido crearPartido(Jugador jugadorA, Jugador jugadorB, Mazo
mazo) {
        return new Poker(jugadorA,jugadorB,mazo);

    }
}

public class FactoryTruco extends FactoryPartido {

    public FactoryTruco() {
        super();
    }

    public Jugador crearJugador() {
        return new JugadorTruco();
    }

    public Mazo crearMazo() {
        return new BarajaEspañola();
    }

    public Partido crearPartido(Jugador jugadorA, Jugador jugadorB, Mazo
mazo) {
        return new Truco(jugadorA,jugadorB,mazo);
    }

}
```

- AbstractFactory, subclasea cada familia de productos que desea crear.
- Generalmente las Factories concretas son Singleton
- Los métodos crearJugador(), crearMazo(), crearPartido() responden al Pattern: ***FACTORY METHOD*.**

Paso de
`new JugadorTruco()` → no hay objeto receptor, es una palabra reservada para el compilador
a
`factory.crearJugador()` → se envía un mensaje al objeto responsable de crear los jugadores (hay polimorfismo en la instanciación de los objetos)

- La idea es que encapsulando en un método la creación de instancias, podemos redefinir en las subclases este método, permitiendo a aquel que diseña las subclases cambiar el tipo de clase concreta a crear.

Una variante al ejercicio de **Abstract Factory** es hacer 1 factory único para no subclasificar. Me queda una única clase



¿qué gano? Aíslo la creación en una sola clase.
¿qué pierdo? ¡el polimorfismo!

Surge un problema, si hay por ejemplo 20 tipos de juego de cartas. Sería engorroso crear subclases para cada tipo de juego.

Pero podríamos implementarlo, sin subclasear, aplicando otro Pattern:

PROTOTYPE

```
FactoryPartido factory=new PrototypicalFactoryPartido(
                           new JugadorTruco(),
                           new BarajaEspañola()
                           );
Final finalTorneo = new Final();
finalTorneo.crearPartidos(factory);
finalTorneo.jugar();
```

Estamos configurando al factory con las instancias que queremos que utilice, para que luego el factory, en vez de instanciar, CLONE estas instancias.

```
public class PrototypicalFactoryPartido extends FactoryPartido{

   private Jugador jugador;
   private Mazo mazo;


   public PrototypicalFactoryPartido(Jugador jugador,Mazo mazo) {
      this.jugador=jugador;
      this.mazo=mazo;
   }

   public Jugador crearJugador(){
      return (Jugador)this.jugador.clone();
   }

   public Mazo crearMazo(){
      return (Mazo)this.mazo.clone();
   }

   public Partido crearPartido(Jugador jugadorA,Jugador jugadorB, Mazo
mazo){
      return new Partido(jugadorA,jugadorB,mazo);
   }

}
```

Para poder clonar Jugadores y Mazos, esos objetos deben implementar la interfaz CLONEABLE.

public Object clone()

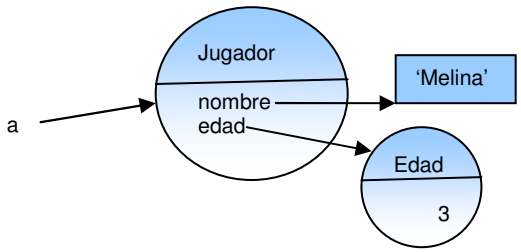
Si quisiéramos clonar un objeto que no implemente esta interfaz, esto arrojaría una CloneNotSupportedException.

Object.clone() es un metodo de scope protected, y cuando queremos indicar que se puede hacer una copia, atributo a atributo, de un objeto, implementamos la interfaz, y así permitimos a los demas objetos que lo clonen.
Object.clone() hace una Shallow Copy (Copia superficial) del objeto, porque primero lo copia, y después va pasando todos los atributos del objeto original a la copia.

Copia superficial y copia profunda

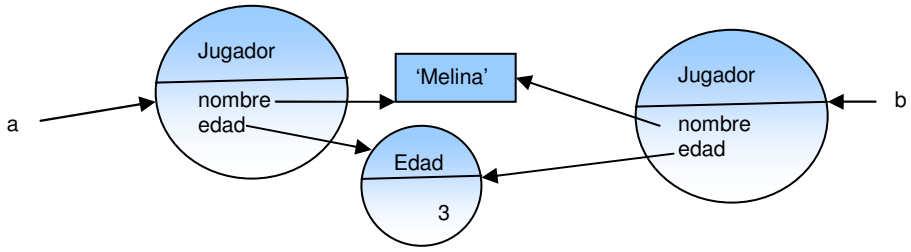
Shallow copy

Copio el objeto original y conservo las mismas referencias para sus atributos.
Si tenemos:




```
Jugador b = (Jugador) a.clone();
```

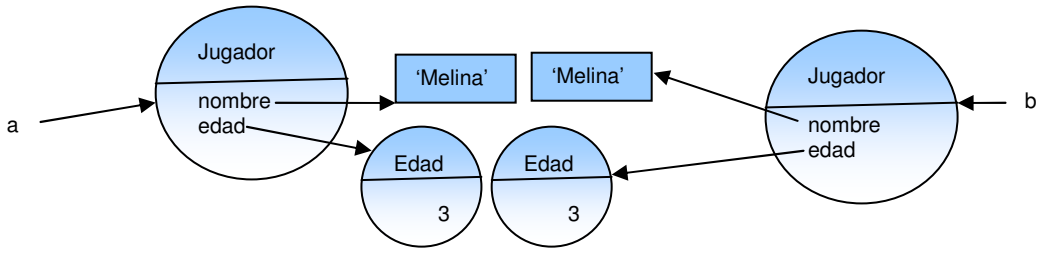
Nos queda:



Deep copy

Se generan nuevas copias del objeto original y de sus atributos (en forma recursiva, anidando los n niveles de profundidad; el problema es identificar referencias circulares entre objetos para no ciclar indefinidamente).

```
Jugador b = (Jugador) a.deepClone();
```
(o algo similar)



Nota: no siempre es posible obtener una nueva copia de todos los objetos.
Para más detalles sobre clonación, ver la documentación del método.

Un ejemplo de cómo implementar el clone en Jugador:

```
public class Jugador implements Cloneable {

      …

      public Object clone() throws CloneNotSupportedException {
            return super.clone();
      }
```

Puedo usar el de Object, sólo debo implementar Cloneable, lo que está bueno porque en tiempo de desarrollo puedo saber qué objetos puedo clonar y cuáles no (puede ser considerado una ventaja respecto de St, donde si quiero restringir que un objeto pueda clonarse tengo que implementar un método a mano para que tire error, y así evitar que lo herede de Object:

```
copy (equivalente al clone() de Java)
      self error: 'Conmigo no podés, lo siento'
```

Igual, la ventaja que tengo es que en St puedo cambiar quiénes se clonan y quiénes no en forma dinámica).

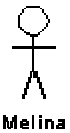
El PROTOTYPE es imprescindible en ambientes como Self, donde no tengo clases (para poder extender funcionalidad). En lenguajes como C++/Java es muy útil porque así "genero" clases dinámicamente (objetos en lugar de clases, con la restricción de que deben tener el mismo estado –no funcionaría si un jugador de tute tiene como atributo sólo nombre y el de truco sólo edad-). En Smalltalk no es tan necesario porque como las clases son objetos, ya existe la idea cuando hago:

```
Object subclass: #Animal
      instanceVariableNames: 'name'
      classVariableNames: 'KnowledgeBase'
      poolDictionaries: ''
      classInstanceVariableNames: ''
```

¿Cuándo conviene usar PROTOTYPE?

Si la instanciación de un objeto es costosa (consume recursos críticos, ya sea acceso a base de datos y posterior conversión a objetos, o un archivo serializado, etc. etc.) y además los objetos tienen un estado poco variable (las cartas son un muy buen ejemplo, serían objetos inmutables).

Dentro del ejemplo, podemos preguntar qué pasaría si el jugador tiene un conjunto de bitmaps para dibujarse en la pantalla y una etiqueta con el nombre.



Si todos los jugadores comparten el mismo bitmap y sólo cambian el Label, estaría bueno no tener que redibujar el gráfico del jugador cada vez que instancie uno; en lugar de eso se podría clonar el bitmap y reemplazar solamente la parte del Label. Entonces

lo que hacemos es un método adjustClone(String nombre) para que después de clonar el objeto le cambie el nombre, algo como:

#PrototypicalFactoryPartido:

```
	public Jugador crearJugador(String nombre) {
		Jugador jugador = null;
		try {
			jugador = (Jugador) this.jugador.clone();
		} catch (CloneNotSupportedException e) {
			... bla ...
		}
		jugador.setNombre(nombre);
		return jugador;
	}
```

Y desde el cliente:

```
FactoryPartido trucoFactory =
     new PrototypicalFactoryPartido(new JugadorTruco(), new Mazo());

Jugador luli = trucoFactory.crearJugador("Melina");
```

Volvamos sobre la implementación del Abstract Factory:

```
Jugador jugadorA=factory.crearJugador();
Jugador jugadorB=factory.crearJugador();
Mazo mazo1=factory.crearMazo();
Partido partido1=factory.crearPartido(jugadorA,jugadorB,mazo1);
```

Al hacer esto, de alguna forma estamos al tanto de la representación interna de Partido, estamos al tanto de cómo componer el objeto, y si esta composición fuera compleja, no nos sería útil usar esta solución.

El pattern:
BUILDER nos acerca otra alternativa…

```
   public void crearPartidos(PartidoBuilder builder){

      builder.buildJugador();
      builder.buildJugador();
      builder.buildMazo();
      partidos.add(builder.getPartido());

      builder.buildJugador();
      builder.buildJugador();
      builder.buildMazo();
      partidos.add(builder.getPartido());

   }

public class PartidoBuilder {

   ArrayList jugadores;
   Mazo mazo;

   public PartidoBuilder(){
      jugadores=new ArrayList();
   }

   public void buildJugador(){
     this.jugadores.add(new Jugador());
   }

   public void buildMazo(){
      this.mazo=new Mazo();
   }

   public Partido getPartido(){
      return new
Partido((Jugador)jugadores.get(0),(Jugador)jugadores.get(1),mazo);
   }
}
```

La idea del Builder es:

Por un lado, pedirle que vaya armando las partes que componen el objeto, sin saber de qué clases son, y además pedirle el producto final, sin saber cómo se ensamblan las partes que componen el objeto.
A medida que le vamos indicando que cree las partes, éstas se guardan en atributos del Builder, o sea, en su estado.

Builder es STATEFUL
Factory es STATELESS

Algunas observaciones sobre el BUILDER:

- Si tengo que agregar referís, y más elementos ("productos") también tengo que cambiar la interfaz, al igual que en el Abstract Factory. Se agrega un buildReferi();
- El Builder presenta una ventaja comparativa si la estructura de los objetos que construyo cambia de un producto a otro. Supongamos un Partido de Tute, donde necesito al menos 3 jugadores:

En un Abstract Factory tengo que cambiar la firma del método crearPartido:
`crearPartido(Jugador, Jugador, Jugador, Mazo)`
con lo que cambia la interfaz y tengo que ver el impacto en todos los clientes.

En cambio con el Builder sólo necesitamos hacer:

```
builder.buildJugador();
builder.buildJugador();
builder.buildJugador();
builder.getPartido();
```

- El Builder asegura que el producto final está construido correctamente: el getPartido() (deberían considerarse que si no se instanciaron el mazo y los jugadores el objeto Partido no está correctamente construido):

```
public Partido getPartido(){
      if (jugadores.size() < 2) {
            throw new FaltanJugadoresException();
      }
      if (mazo == null) {
            throw new FaltaMazoException();
      }
      return new Partido((Jugador)jugadores.get(0),
                         (Jugador)jugadores.get(1),
                          mazo);
   }
```

Breve ejemplo de Singleton

Intención del Singleton: para una clase determinada queremos tener:

- una sola instancia en el ambiente
- un único punto de acceso en el sistema

Lo implementamos a través de una variable de instancia que se inicializa en forma tardía la primera vez que lo pide un usuario (por el getInstance()), y dejando el constructor privado:

```
public class SingletonExample {

	private static SingletonExample instance = null;

	private SingletonExample() {}

	static public SingletonExample getInstance() {
		if (instance == null) {
			instance = new SingletonExample();
		}
		return instance;
	}
}
```

Para investigar:

- ¿Qué pasa si subclasifico SingletonExample?
- ¿Qué pasa si tengo muchas Virtual Machines?

Otra variante es declarar la clase final y tener solamente métodos static:

```
public final class PrintSpooler
{
	//a static class implementation of Singleton pattern
	static public void print(String s) {
		System.out.println(s);
	}
}
```

- Al declarar la clase como **`final`** evito que la subclasifiquen.
- Con esta técnica es costoso permitir luego tener varias instancias.
- Los objetos que implementen esta técnica no tienen estado (son stateless), ya que como todos los métodos son static no puedo inicializar variables.

Resumen

Patterns Creacionales:

1. **Abstract Factory**
2. **Factory Method**
3. **Prototype**
4. **Builder**
5. **Singleton**

- Abstraen el proceso de instanciación.
- Nos ayudan a lograr que un sistema se independice de cómo sus objetos se crean, componen y representan.

Hay dos puntos fundamentales en estos Patterns:

- Encapsulan la información acerca de que clase concreta usa el sistema.
- Esconde la forma en que estas instancias son creadas y compuestas.

FACTORY METHOD

- Define una interfaz para crear objetos,

Si el objeto Factory es abstracto, deja a las subclases la instanciación.
Si el objeto Factory es concreto, le da la posibilidad de delegar la instanciación a sus subclases, agregando flexibilidad al modelo, ya que los que se ocupen de implementar las subclases, podrían cambiar el tipo concreto de los objetos creados.

Ventajas

- Elimina la necesidad de indicar las clases concretas en el código

Desventajas

- Los clientes tienen que subclasear al Objeto Factory, por cada tipo de producto que deseen crear.

ABSTRACT FACTORY

- Provee una interfaz para crear familias de objetos relacionados, sin especificar su clase concreta.

Esa interfaz que provee, es un conjunto de Factory Methods.

Ventajas

- Aísla clases concretas
- Hace fácil intercambiar el tipo de productos creados,.
- Consistencia entre productos, porque creamos un conjunto de objetos de un mismo tipo.

Desventajas

- Para agregar un nuevo tipo de producto, hay que cambiar la interfaz del Factory y sus subclases

Las Factories concretas, generalmente se implementan con un SINGLETON.

PROTOTYPE

- Usa una instancia prototipo, para especificar el tipo de producto a crear. Y crea nuevos objetos copiando este prototipo.

Ventajas

- Poder cambiar el prototipo en tiempo de ejecución
- Evitar subclases en Factory Method

Desventajas

- Tener que implementar clone()

BUILDER

- Separa el proceso de construcción de un objeto(paso a paso), de su representación interna(forma en que sus partes están ensambladas)

Ventajas

- Podemos variar la forma de ensamblar las partes que componen el objeto, subclaseando el builder.
- Su simplicidad, aísla el algoritmo de armado del objeto y las Clases de las partes que forman el objeto.

Desventajas

- Su rigidez, como el algoritmo de armado esta aislado, tenemos que adaptarnos a esa representación interna del objeto.

Bibliografía

- *Design Patterns,* Erich Gamma et al., Addison-Wesley (referido como DP)
- *The Design Patterns Smalltalk Companion*, Sherman Alpert, Kyle Brown, Bobby Wolf, Addison-Wesley.
- *The Design Patterns Java Companion*, James Cooper, Addison-Wesley.