Patrones Creacionales

1 Introducción

1.1 Versiones

Autores principales	Versión	Fecha	Observaciones
Gastón Prieto, Franco Bulgarelli	1.1	2020	Ejemplos de código actualizados de Smalltalk a Pseudocódigo
Carla Griggio y Germán Leiva	1.0	2012	Versión original

1.2 ¿Qué es eso?

Los patrones creacionales son estrategias en el paradigma de objetos que sirven para abstraer el proceso de construcción de de los mismos, particularmente cuando este proceso de configuración es complejo, tedioso o difícil de mantener. Según GoF[1]:

(...) ayudan a hacer que nuestro sistema se independice de cómo sus objetos son creados, compuestos y representados.

Pero, ¿por qué la creación de los objetos podría complicarse? ¿Crear no se trata de simplemente hacerle new a la clase que queremos instanciar? ¿Crear, instanciar, configurar es todo lo mismo? Para responder a esas preguntas, nos va a convenir hacer algunas diferenciaciones.

1.3 Configuración vs. Instanciación

Cuando tenemos objetos sencillos, como una pequeña golondrina voladora que transforma alpiste en energía...

```
clase Golondrina
   constructor
       this.energia = 100

   metodo comerAlpiste()
       this.energia += 10

   metodo volar()
      this.energia -= 10
```

[1] **Design Patterns - Elements of Reusable Object-Oriented Software**
Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides / Addison-Wesley 1995
GoF = Gang of Four (en referencia a los 4 autores)

... obtener un ***objeto listo para ser usado*** es trivial:

```
pepita = new Golondrina()
```

¿Y por qué? Sencillamente porque no hay nada que *configurar*, es decir, no hay ningún *estado inicial* que al usarla tengamos que darle a pepita: cuando es creada su energía inicia en 100. En otras palabras, crear a pepita se reduce a una simple instanciación, en otras palabras ***traer al mundo de los objetos uno nuevo de la mano de su clase***.

Podríamos dar un poco más de libertad a quienes usen la clase Golondrina, y permitir especificar la energía inicial:

```
pepita = new Golondrina(100)
anastasia = new Golondrina(250)
```

Esto nos da más flexibilidad, pero también hace la *configuración* más compleja: ahora **es nuestra responsabilidad** dar un nivel (válido) de energía inicial. Incluso podríamos dar más flexibilidad y llevar la *inicialización* de la golondrina a *setters*:

```
anastasia = new Golondrina()
anastasia.setEnergia(250)
```

Nuevamente ahora ganamos la posibilidad de elegir ***cuándo*** definir ese estado inicial, pero también es más propenso a error: ¿qué pasará si nos olvidamos de enviar `setEnergia`?

```
anastasia = new Golondrina()
anastasia.volar()
```

Quizás volar lance una excepción de tipo `NullPointerException`, `NoMethodError`, `NilDoesNotUnderstand`, etc. O quizás funcione, asumiendo una energía inicial de 100, o 0. ¡Quién sabe! En otras palabras, probablemente **habremos perdido robustez**.

Con esto en mente, ya podemos dar nuestras primeras definiciones:

- **instanciación:** es el acto de, a partir de una clase, traer al ambiente un nuevo objeto. Ni más, ni menos.
- **inicialización**: es el acto de darle valor inicial al estado del objeto. Esto puede ocurrir enviando mensajes de tipo *setter*, pasando parámetros por constructor, o combinando estas ideas y variantes.
- **construcción, creación o configuración** (usaremos estos términos de forma más o menos indistinta): es el proceso que abarca las dos ideas anteriores. En otras palabras, el ***proceso creacional trata de traer un objeto al ambiente y darle un estado inicial consistente y conocido, que tenga sentido en el dominio y que lo deje "listo para usar"***

Si bien tanto la inicialización como la instanciación en sí mismas son tareas sencillas, basadas en operaciones primitivas del lenguaje como las asignaciones y el uso del `new`, la configuración es un proceso que puede tornase largo y complicado cuando tenemos... ¡objetos complejos!

1.4 Algunos ejemplos de creaciones "complicadas"

Strategy

```
Planificador planificador = new Planificador();
planificador.setEstrategia(new FIFO());
List<Proceso> procesos = new ArrayList<Proceso>();
procesos.add(new Proceso(1412,"kernel"));
procesos.add(new Proceso(1413,"demon"));
procesos.add(new Proceso(1414,"keyLogger"));
...
planificador.setProcesos(procesos);
```

En este ejemplo podemos ver que para que un planificador tenga sentido es necesario asignarle una estrategia (un planificador está compuesto por 2 objetos, el planificador propiamente dicho y su estrategia). Se podría haber optado por utilizar un constructor e instanciar el planificador de la siguiente manera

```
Planificador planificador = new Planificador(new FIFO());
```

De todas formas podemos ver que no es una "mera instanciación". Al utilizar composición la construcción de este objeto requiere tomar algunas decisiones de diseño.

1.4.2 Decorator

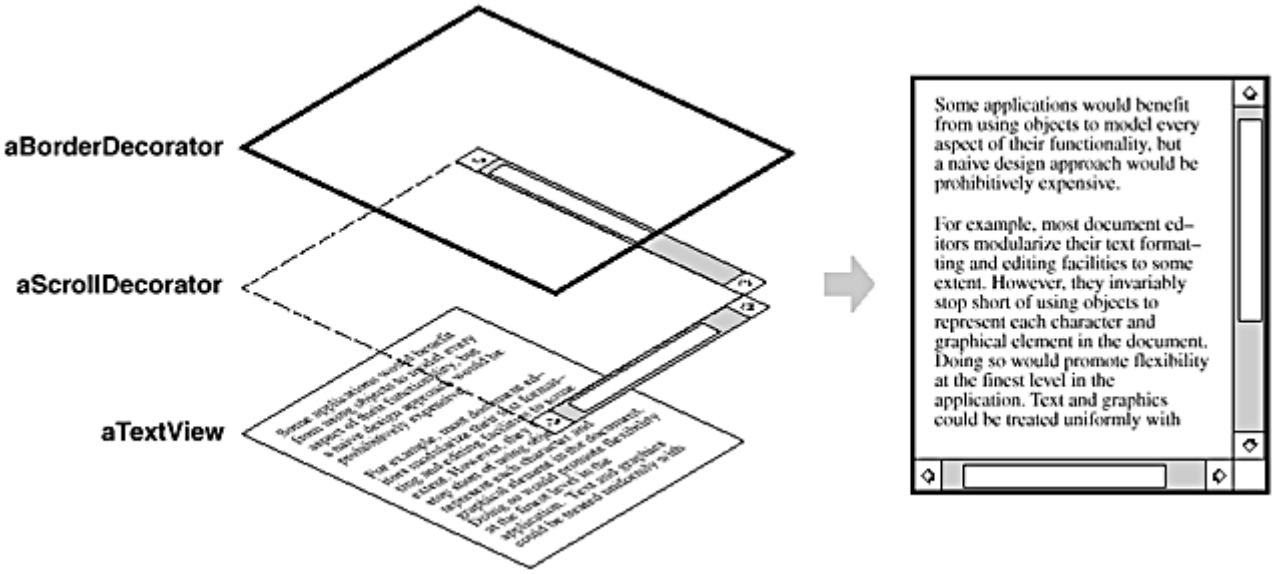
El patrón decorator[2] es una solución compleja a un problema también complejo. Para nuestro estudio creacional no es relevante cómo se comporta, pero sí que su construcción no es para nada trivial:

```
Widget[3] text = new Texto("Hola soy un texto");

Widget scrollA = new Scroll();
Widget bordeB = new Borde();

bordeB.setDecorado(texto);
scrollA.setDecorado(bordeB);

Widget ventanaScroleableConBordeConTexto = scrollA;

scroll2.setDecorado(texto);
borde2.setDecorado(scroll2);

Widget ventanaBordeadaConScrollConTexto = borde2;
```

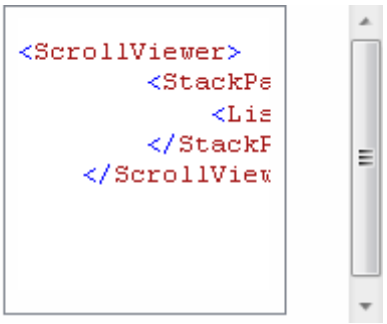
En este ejemplo podemos ver que no solo es importante la instanciación de los objetos por el uso de la composición, sino que además, la forma en que se componen los objetos va a determinar el comportamiento final que vamos a obtener. No es lo mismo una `ventanaScroleableConBordeConTexto` que una `ventanaBordeadaConScrollConTexto`.

---

[2] Para información sobre este patrón, consultar acá
[3] Es un término utilizado para denotar un elemento visual que se va a ver en la pantalla.



Este sería el dibujo de una `ventanaBordeadaConScrollConTexto`



Si le ponen mucho amor... esto sería una `ventanaScroleableConBordeConTexto`

1.5 Algunas generalidades más

Es importante identificar cuando nuestra solución a la hora de construir objetos no está libre de duplicaciones, no es simple, intuitiva o está muy acoplada al objeto que la utiliza. En ese sentido los patrones creacionales no nos van a dar grandes nuevos conceptos, sino que son una particularización de otras ideas y cualidades de diseño, aplicadas a la creación de los objetos.

Por ejemplo, si cuando vemos a dos objetos que se envíen mensajes nos preocupa no romper su encapsulamiento, desde el punto de vista creacional también nos interesará no hacerlo cuando uno construye a otro. O de igual forma, si procuramos validar entradas para que en el un uso erróneo de un objeto falle rápido, también buscaremos que una configuración incorrecta sea reportada rápidamente y no nos deje en nuestras manos un objeto inconsistente.

De todas formas, cuando estudiemos a los objetos desde el punto de vista creacional, **no haremos foco en las cuestiones de comportamiento**: como vimos en los ejemplos anteriores y notaremos en los que vendrán, ni siquiera nos tomaremos el trabajo de mostrar los métodos de instancia de los objetos que estamos configurando, porque **sólo nos preocuparemos por su construcción**.

2 Algunos patrones fundamentales

2.1 Singleton

2.1.1 Intención

El patrón singleton tiene dos objetivos íntimamente relacionados:

1. asegurarse que una clase tenga una única instancia
2. y proveer un único punto de acceso global a ella.

2.1.2 ¿Cuándo se puede usar?

Usaremo al patrón singleton cuando deba haber una única instancia de una clase en **todo el sistema** y deba ser accesible a los clientes (es decir, a los objetos que la usan) desde un punto de acceso **global** bien conocido.

Dado que estas situaciones no son tan frecuentes y, como veremos a continuación, su uso trae bastantes desventajas, es un patrón que usaremos muy poco y sólo en aquellos casos en que **no nos quede otra opción**.

2.1.3 Ejemplo

En el mundo de Dragon Ball, existen 7 esferas, que una vez que se juntan, nos permiten invocar al dragón Sheng Long y pedirle un deseo, que sigue ciertas reglas que (dado que estamos hablando de problemas creacionales) no nos interesan.

En esta situación el dominio mismo es el que nos lleva a pensar en singleton, dado que no tendría sentido que hubiera más de un dragón Sheng Long. Además:

- Quiero mantener sólo un estado interno
- No quiero que ese estado se repita.
- Quiero tener sólo 1 punto de acceso: por lo general, a esa instancia quiero accederla de sólo una manera.

En otras palabras, en este problema Sheng Long será la única instancia del DragonQueConcedeDeseos:

```
// hasta acá es una clase común y corriente
clase DragonQueConcedeDeseos
   constructor
     ...inicializamos al dragón...

    metodo concederDeseo(List<Esfera> esferas, Deseo deseo)
      ...concedemos los deseos...
```

El problema que nos encontramos es que si bien esta línea funciona...

```
shengLong = new DragonQueConcedeDeseos()
```

....es semánticamente incorrecta. Es decir, quien quiera hacer eso y no conozca mi necesidad de que haya sólo un objeto dragón, estará esperando una nueva instancia cada vez que utilice `new` (es lo que dice la palabra).

En su lugar, se elige tener un método de clase `instance` o `getInstance` (por convención), mediante el cual puedo acceder de manera unívoca a mi instancia:

```
shengLong = DragonQueConcedeDeseos.instance()
```

Dicha instancia va a estar almacenada en la clase (con una variable de clase ó static), que por convención llamaremos `INSTANCE`:

```
clase DragonQueConcedeDeseos
   // hacemos al atributo final (no modificable) para que nadie
   // lo pueda modificar y privado para que nadie lo pueda acceder externamente
   private static final DragonQueConcedeDeseos INSTANCE = new DragonQueConcedeDeseos();

   metodo static instance()
      retornar  INSTANCE

   // hacemos el constructor privado, para que nadie más lo puede llamar
   private constructor
      ...inicializamos al dragón...
```

Con estos cambios, un posible objeto cliente lo usaría así:

```
 // Cualquier otro componente que use al dragón
class GuerreroZ
   void cumplirDeseo()
      List<Esfera> esferas = recolectarEsferas()
      Deseo deseo = pensarDeseo()
      DragonQueConcedeDeseos.instance().concederDeseo(esferas, deseo)
```

Con esto, si alguien intenta crear nuevas instancias o acceder al atributo, fallará:

```
// no compila
otroDragon = new DragonQueConcedeDeseos();
// no compila
DragonQueConcedeDeseos.INSTANCE
```

Vale aclarar que el uso de finals y privates pueden verse como programación defensiva o incluso deformaciones del patrón de diseño Singleton. Solamente se mencionan aquí porque son prácticas estándares y sencillas de lenguajes como Java y para dar un pantallazo, pero remarcamos que la intención del patrón no es asegurar la "seguridad del acceso" y que esa responsabilidad debería estar a otro nivel ya que casi siempre existe una forma de violar estas protecciones desde el propio lenguaje.

En otras palabras, esta también sería una definición válida de un singleton

```
clase DragonQueConcedeDeseos
```

```
   // de clase, pero no private ni final
   static DragonQueConcedeDeseos INSTANCE = new DragonQueConcedeDeseos();

   metodo static instance()
      retornar  INSTANCE

   // dejamos un constructor común y corriente, como al inicio
   constructor
      ...inicializamos al dragón...

// compila, pero NO lo haremos porque consensuamos no hacerlo
otroDragon = new DragonQueConcedeDeseos();
// compila, pero NO lo haremos porque consensuamos no hacerlo
DragonQueConcedeDeseos.INSTANCE
```

2.1.4 Problemas

Como se ve, es muy fácil hacer que una clase sea *singleton*, pero ¿es una buena idea? Supongamos que tenemos una clase Recomendador, para sugerir la construcción de atuendos a partir de diferentes prendas, inspirado en el dominio de Que Me Pongo y decidimos hacerlo singleton:

```
clase Recomendador
  Estacion estacion;

  metodo configurarEstacion(Estacion estacion)
    this.estacion = estacion

  Atuendo recomendar(List<Prenda> prendasPosibles)
    ....
    retornar new Atuendo(...)
```

¿Cómo podríamos testear este componente? Imaginemos este test:

```
    test si_le_doy_un_pantalon_y_estoy_en_verano_prefiere_ropa_corta()
      Recomendador.instance().configurarEstacion(Estacion.VERANO)
      assert Recomendador.instance().recomendar(....).contains(....)

    test si_le_doy_un_pantalon_y_estoy_en_invierno_prefiere_ropa_larga()
      Recomendador.instance().configurarEstacion(Estacion.INVIERNO)
      assert Recomendador.instance().recomendar(..).contains(....)
```

Estos tests simplemente configuran un recomendador, cada uno para una estación diferente, y luego lo usan y validan los resultados. Pero ¿qué pasa si nos olvidamos de configurar el recomendador?

```
    test otro_test_en_que_me_olvide_de_configurar_la_estacion()
      assert Recomendador.instance().recomendar(....).contains(....)
      // ¡*****andá a saber que pasa aca*****, depende del orden!
```

¿Cómo debería comportarse el recomendador? ¿Como uno no configurado? ¿Como uno configurado en el último test? ¿Y cual es el último test?

¡La verdad es que no lo sabemos! Muchos frameworks de testing no garantizan que los tests se ejecuten en un orden particular, y otros incluso por defecto los corren, a propósito, en un orden al a azar, para asegurar la independencia de los test entre sí.

Pero **lo que sí sabemos** es el recomendador que usaremos en todos los tests es el mismo, porque al ser singleton existirá uno solo en toda la máquina virtual. Y eso puede llegar a dar problemas, porque los tests pasan a compartir un estado global, con consecuencias no siempre controlables.

Veamos luego otra situación: queremos un método (en alguna clase) que compare dos recomendaciones:

```
boolean recomiendaLoMismo(estacion1, estacion2, prendas)
  Recomendador.instance().configurarEstacion(estacion1)
  Recomendador.instance().configurarEstacion(estacion2)

 return Recomendador.instance().recomendar(prendas) == Recomendador.instance().recomendar(prendas)
```

Salta a la vista que este código no tiene sentido (una configuración pisa la otra), y para que funcione correctamente deberíamos reordenar e introducir variables locales:

```
  boolean recomiendaLoMismo(estacion1, estacion2, prendas)
    Recomendador.instance().configurarEstacion(estacion1)
    resultado1 = Recomendador.instance().recomendar(prendas)

    Recomendador.instance().configurarEstacion(estacion2)
    return resultado1 == Recomendador.instance().recomendar(prendas)
```

¡Ough! Si bien funciona[4], el código de volvió bastante complicado.

En todos los casos podríamos corregir el problema teniendo más cuidado o reordenando el código, pero es más fácil si tenemos múltiples instancias:

```
// Problema 1

test si_le_doy_un_pantalon_y_estoy_en_invierno_prefiere_ropa_larga()
  Recomendador recomendador = new Recomendador(Estacion.INVIERNO)
  assert recomendador.recomendar(....).contains(....)

test si_le_doy_un_pantalon_y_estoy_en_verano_prefiere_ropa_corta()
  Recomendador recomendador = new Recomendador(Estacion.VERANO)
  assert recomendador.recomendar(....).contains(....)

test otro_test_en_que_me_olvide_de_configurar_la_estacion()
  // ¡ni compila!
  assert recomendador.recomendar(....).contains(....)
```

[4] ¡Siempre y cuando no tengamos múltiples hilos (threads)!

```
// Problema 2

boolean recomiendaLoMismo(estacion1, estacion2, prendas)
  // ¡no hay ambigüedad!
  return new Recomendador(estacion1).recomendar(prendas)
      == new Recomendador(estacion2).recomendar(prendas)
```

Como vemos, la solución a los problemas de código que nos puede traer un singleton es zen: *en lugar de corregir al singleton, es más fácil y seguro no usar singleton*. Y como se ve en los ejemplos, esto es particularmente cierto cuando éste tiene **estado mutable.**

2.2 Factory Method

2.2.1 Intención

La intención del patrón Factory Method es definir una **interfaz** para crear **un** objeto, pero deja a las subclases decidir qué clase concreta instanciar y cómo. Le permite a una clase "delegar" la instanciación o parte de la creación a sus subclases.

2.2.2 Ejemplo

Estamos desarrollando un sistema para administrar un emprendimiento de Pizza Party: le proveemos a personas y empresas pizzas para sus eventos. En nuestras primeras iteraciones diseñamos un objeto `FabricaDePizza`, al que podemos decirle que genere una colección de pizzas para una cierta cantidad de personas:

```
// ¡Pizza party!
fabrica = new FabricaDePizza()
fabrica.setCantidadDePersonas(400)
pizzas = fabrica.fabricarPizzas()
```

La implementación de esta clase es bastante directa: vamos a crear una cierta cantidad de pizzas en función de la cantidad de comensales:

```
clase FabricaDePizza
  int cantidadPersonas

  List<Pizza> fabricarPizzas()
    retornar (1..estimarCantidadDePizzas()).map((it) => fabricarPizza())

  int estimarCantidadDePizzas()
    retornar ...

  // los objetos pizza nos servirán para llevar estadísticas,
  // estimaciones de costos, ingredientes,
```

```
  // tiempos y otras cuestiones de las operaciones de la empresa.
  Pizza fabricarPizza()
    pizza = new Pizza()
    pizza.preparar()
    pizza.cocinar()
    pizza.cortar()
    pizza.empaquetar()
    retornar pizza
```

Como se observa, una cosa es instanciar una pizza (`new Pizza()`), otra cosa es *crearla* o *construirla* (`preparar/cocinar/cortar/empaquetar`).

Sin embargo, por ese lado todo marcha de maravilla. La complicación viene por el lado creacional: actualmente queremos transicionar al siguiente diseño, que nos permita tener diferentes *"sabores"* de fábricas de pizza....

```
// complejizando las cosas
fabrica = new FabricaDePizzaAptoVeganos()
// o también...
// fabrica = new FabricaDePizzaEstandar()
// fabrica = new FabricaDePizzaAptoCeliacos()
// etc...
fabrica.setCantidadDePersonas(400)
fabrica.fabricarPizzas()
```

...que siga básicamente el mismo algoritmo...

```
class FabricaDePizzaX
  Pizza fabricarPizza()
    pizza = ....crear una pizza propia  de la fábrica X....
    pizza.preparar()
    pizza.cocinar()
    pizza.cortar()
    pizza.empaquetar()
    return pizza
```

...pero que cada una cree un *tipo* de pizza diferente. Acá diremos *tipo* en sentido amplio: podrían ser instancias de clases diferentes, o de la misma clase pero con configuraciones totalmente distintas.

Entonces algo que podemos hacer es convertir a nuestra `FabricaDePizza` en abstracta e introducir un método abstracto `crearPizza`:

```
abstract class FabricaDePizza
  Pizza fabricarPizza()
    pizza = this.crearPizza()
    pizza.preparar()
    pizza.cocinar()
    pizza.cortar()
    pizza.empaquetar()
```

```
    return pizza

  abstract Pizza crearPizza()
```

De esta forma podríamos tener diferentes implementaciones...

```
class FabricaDePizzaEstandar extends FabricaDePizza
  Pizza crearPizza()
    return new PizzaMuzzarela()

class FabricaDePizzaAptoVeganos extends FabricaDePizza
  Pizza crearPizza()
    return new PizzaVegana()
```

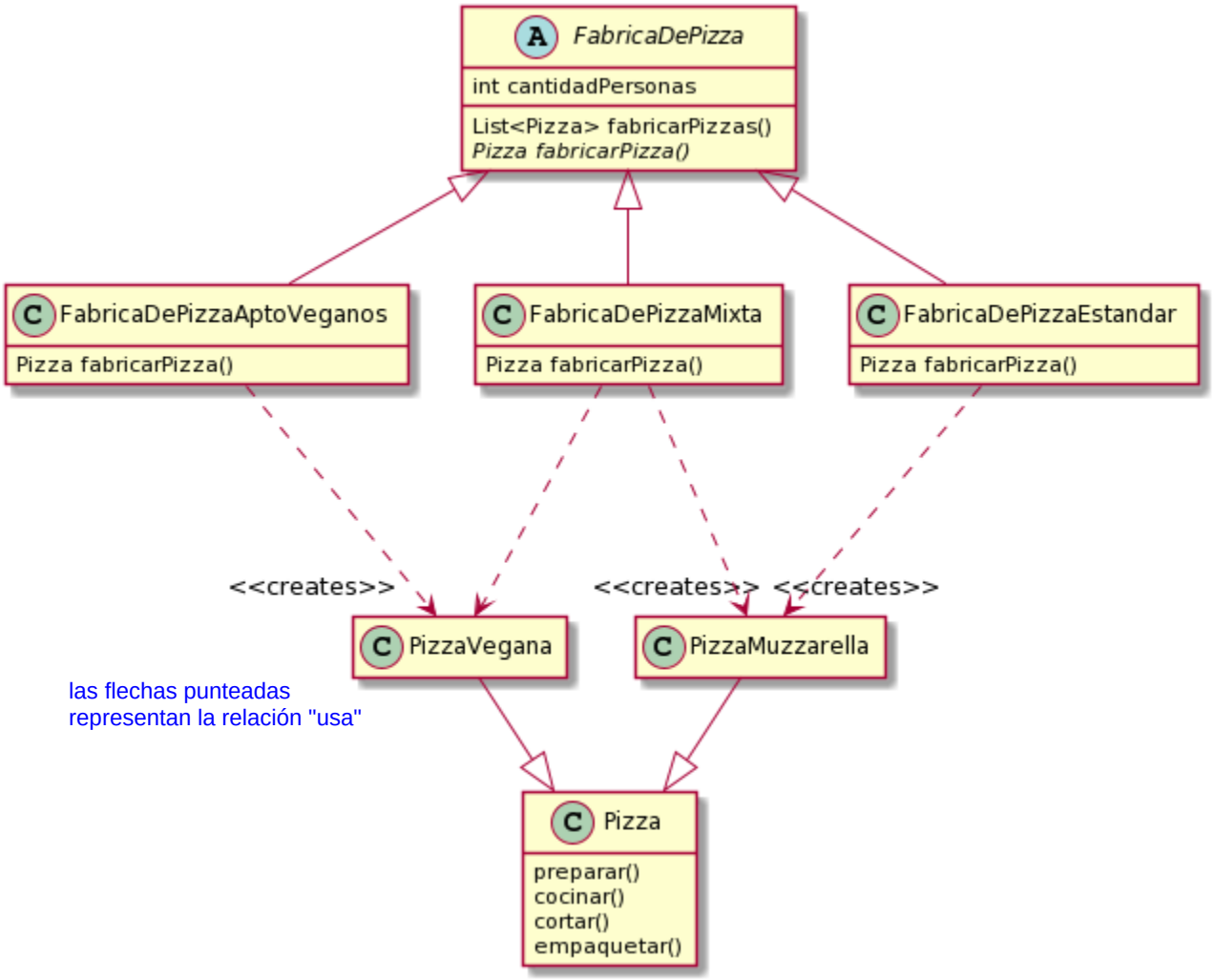
...en las que cada una decida la clase concreta que se va a instanciar. Incluso podríamos ir más allá y devolver instancias de la misma clase, pero inicializadas de forma diferente...

```
class FabricaDePizzaPersonalizada extends FabricaDePizza
  constructor(aditamentos)
    this.aditamentos = aditamentos

  Pizza crearPizza()
    pizza = new PizzaMuzzarela()
    pizza.agregarAditamentos(aditamentos)
    return pizza
```

...o tener lógicas aún más complejas:

```
class FabricaDePizzaMixta extends FabricaDePizza
  Pizza crearPizza()
    if (math.random() > 0.5)
      return new PizzaMuzzarela()
    else
      return new PizzaVegana()
```

En el ejemplo, nuestro factory method es `crearPizza`.



Algunas cosas a remarcar:

- En el ejemplo "la clase Factory" es FabricaDePizza.
- Si "la clase Factory" es abstracta, deja a las subclases la instanciación.
- Si "la clase Factory" es concreta, 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 a "la clase Factory" por cada *tipo* de producto que deseen crear

2.2.3 Template Methods y Hooks Methods

Recordando la intención del template method

Template Method: Permitir que ciertos pasos de un algoritmo definido en una operación de una superclase, sean redefinidos en las subclases sin necesidad de tener que sobrescribir la operación entera.

Por ejemplo

```
clase Cuerpo
      double masa

      double densidad()
            return this.getMasa() / this.getVolumen();

clase Cubo hereda Cuerpo
      double lado
      double getVolumen()
            return Math.pow(lado,3);

clase Cilindro hereda Cuerpo
      double altura;
      double radio;

      double getVolumen()
            return Math.pow(radio,2) * 3.14 * altura;
```

En este ejemplo podemos ver que el template-method es el método `densidad()` y los pasos que delega en sus subclases (en este caso es uno solo) es la responsabilidad de calcular el volumen con `getVolumen()`.

A los pasos que deben ser definidos en un template-method también se los conoce como **Hook-Methods**.

> Es común que un **Factory-Method** sea un Hook-Method de algún Template-Method

En el ejemplo de la pizzería

- Template-Method = clase abstract **`Pizzeria`** + `metodo` **`damePizza`**
- Factory-Method = Hook-Method = clase **`Pizzeria`** + `metodo` **`instanciarPizza`** (Responsabilidad de las subclases)

2.2.4 Creation Methods vs Factory Methods

Muchas veces se dice que un factory-method es cualquier método que devuelve una nueva instancia.

Puede ser valioso diferenciar

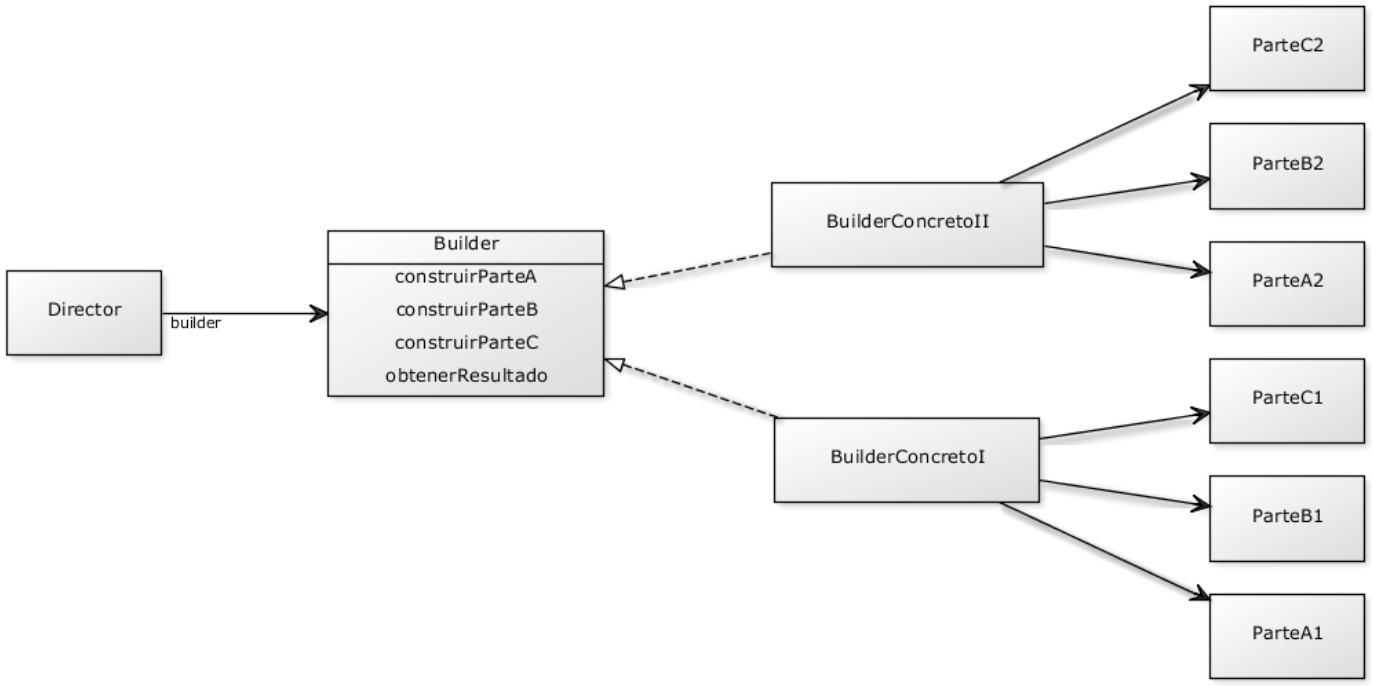
- Factory-method, un patrón de diseño íntimamente relacionado con la herencia de,
- Creation-method, un método cuya única intención es instanciar una clase

2.3 Builder

2.3.1 Intención

Separar la construcción de un objeto complejo de su representación para que el mismo proceso de construcción pueda crear diferentes representaciones.

2.3.2 Estructura



2.3.3 ¿Cuándo se puede usar?

- Cuando el algoritmo para crear un objeto complejo debe ser independiente de las partes que lo forman y de cómo se ensamblan
- Cuando el proceso de construcción permite diferentes representaciones del objeto que se construye

2.3.4 Consecuencias

- Permite variar la representación interna de un producto creando nuevos builders que cumplan con la interfaz correspondiente
- Encapsula el código del ensamblaje de las partes
- Provee un mayor control sobre el proceso de construcción, a diferencia de los patrones creaciones que construyen un objeto de una, el Builder construye el producto paso a paso bajo el control del

director

2.3.5 Ejemplo

Planteamos un ejemplo inocente, hacer café y hacer té.

```
// Por ahora no importa en dónde están estos métodos
public void prepararCafe {
	this.obtenerAguaCasiHirviendo();
	this.molerGranos();
	this.servirEnTaza();
	this.hecharAzucar();
}

public void prepararTe {
	this.hervirAgua();
	this.ponerSaquito(); //Es un té barato
	this.verterAgua();
	this.ponerGotitasDeLimon();
}
```

Al ver que lo que hacen no es tan distinto podríamos estar tentados de lograr algún tipo de polimorfismo entre la construcción de un café y un té.

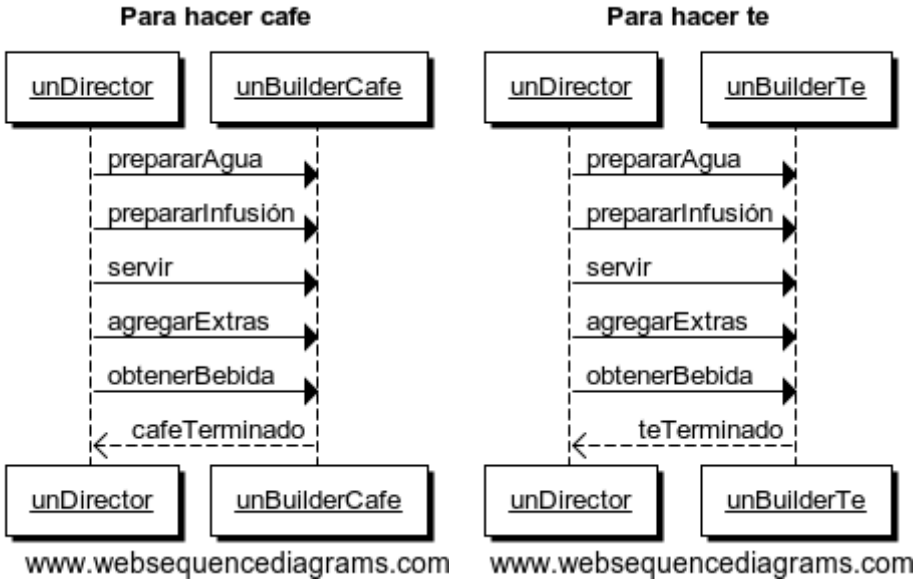
```
public void prepararBebida(String tipoBebida) {
	if (tipoBebida.equals("CAFE")) {
		this.obtenerAguaCasiHirviendo();
		this.molerGranos();
		this.servirEnTaza();
		this.hecharAzucar();
	}
	if (tipoBebida.equals("TE")) {
		this.hervirAgua();
		this.ponerSaquito();
		this.verterAgua();
		this.ponerGotitasDeLimon();
	}
}
```

Si al momento de realizar la construcción sabemos qué producto queremos obtener, un café o un té, podríamos utilizar un Builder para cada caso y con esto no sería necesario parametrizar nada.

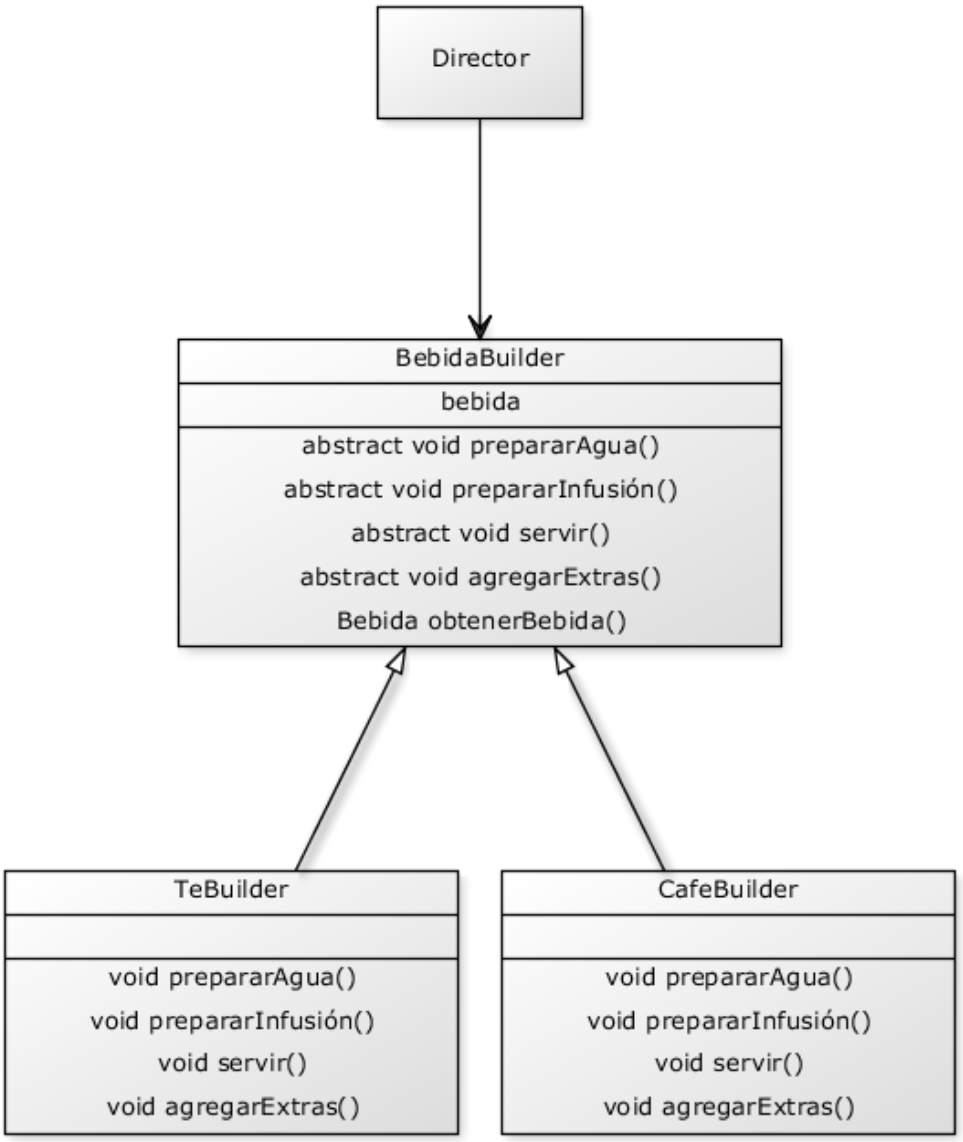
El problema es que los cafés y los tés no tienen el mismo proceso de construcción. Hay varios approach para solucionar esto pero algo sencillo en este ejemplo es que podemos abstraer un proceso de construcción que sí valga tanto para los cafés como para los tés.

```
public abstract class BebidaBuilder {
	public abstract void prepararAgua();
	public abstract void prepararInfusión();
```

```
        public abstract void servir();
        public abstract void agregarExtras();
        public Bebida obtenerBebida();
}
```



Y cada subclase de este builder tendrá su propia implementación de dichos métodos.


Director
BebidaBuilder
bebida
abstract void prepararAgua()
abstract void prepararInfusión()
abstract void servir()
abstract void agregarExtras()
Bebida obtenerBebida()
TeBuilder
void prepararAgua()
void prepararInfusión()
void servir()
void agregarExtras()
CafeBuilder
void prepararAgua()
void prepararInfusión()
void servir()
void agregarExtras()

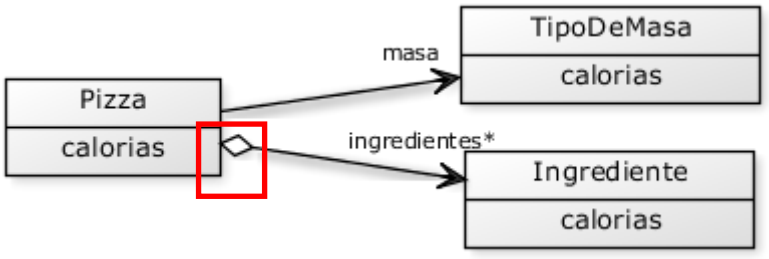
3 Otros patrones y combinaciones más avanzadas

3.1 Builder + Factory Method

Ejemplito inocente.
Queremos hacer pizzas. Todas las pizzas están conformadas de varias partes: masa, salsa, queso, ingrediente principal y aceitunas.
Dependiendo de las partes particulares de las que esté hecha una pizza, para nosotros va a ser un “gusto” en particular y las calorías que tenga van a cambiar.
Las calorías de la pizza van a ser las calorías de la masa + las de sus ingredientes.

Por ejemplo: la Napo tiene masa a la pieda, salsa de tomate, queso muzzarella, tomate y aceitunas verdes.
La fugazzeta rellena tiene masa al molde, no tiene salsa, queso muzzarella, cebolla y aceitunas negras.

Para mí la pizza no deja de ser algo que tiene una masa y muchos ingredientes:



Podemos decir que la creación de determinadas pizzas como la napo o la fugazzeta va a ser frecuente, tienen una representación conocida, y el proceso para crear una pizza mal que mal es el mismo.

Por un lado, pensemos en eso: sea la pizza que sea, cuál sería el proceso de creación de la pizza?

- Agregar masa
- Agregar queso
- Agregar salsa
- Agregar ingrediente principal
- Agregar aceitunas

Para crear una Napo desde un workspace podría hacer:

```
napo = new Pizza().
napo.masa(new ALaPiedra()).
napo.agregarIngrediente(new SalsaDeTomate()).
napo.agregarIngrediente(new Muzzarella())
napo.agregarIngrediente(new Tomate())
napo.agregarIngrediente(new PuñadoDeAceitunasVerdes()).

napo.calorias()
```

OK, eso sirve, pero sigo siendo yo la que sé qué partes tienen que conformar una pizza para que tenga sentido. Nada me impide hacer:

```
new Pizza().calorias()
```

Y que se rompa al pedirle las calorías a su masa. Y aunque no se rompa, tampoco tiene mucho sentido considerar una pizza a algo que es sólo masa.

Eso me lleva a pensar que para tener objetos pizza populando mi sistema que sean consistentes, quiero tratar de evitar crearlos con el “new”, quiero tener maneras de crear una Pizza que me aseguren que vaya a ser consistente.

Una primer idea puede ser trabajar con métodos de clase:

```
napo = Pizza.crearCon(
      new ALaPiedra(),
      new SalsaDeTomate(),
      new Muzzarella(),
      new Tomate(),
      new PuñadoDeAceitunasVerdes()
).
```

Dentro de ese método de clase debería asegurarme que lo que me pasan por parámetro no es fruta (nil cuenta como fruta), porque si la firma del constructor propone recibir cada ingrediente mínimo que necesita la pizza pero por parámetro le pasan otras cosas, nada me asegura que eso vaya a crear una pizza consistente.

Además, qué hay del caso de la Fugazzeta? No lleva salsa, por lo que necesitaría otro constructor.

Y podría pensar lo mismo para otras pizzas: la margherita no lleva muzzarella ni ingrediente principal, sólo Salsa de Tomate y aceitunas. Otro constructor más.

Si yo tuviera que crear una pizza dinámicamente, ¿cómo sé que constructor usar de ante mano? Ese es un problema.
Además, vemos que para distintas pizzas vamos necesitando distintos constructores y hay que ir agrandando la interfaz de Pizza, y agregar más constructores es necesario por la poca flexibilidad de los mismos (es un mensaje y ya).

Hay otro tema: En ningún lugar del modelo queda explícito qué es una pizza napolitana para mí. Los ingredientes de la pizza napolitana son regla de negocio, y estaría bueno que en algún lugar se diga explícitamente qué es una pizza napolitana.

Con métodos de clase se podría hacer:

```
clase Pizza
  metodo de clase nuevaNapo()
      napo = new Pizza()
      napo.setMasa(new ALaPiedra())
      napo.agregarIngrediente(new SalsaDeTomate())
      napo.agregarIngrediente(new Muzzarella())
      napo.agregarIngrediente(new Tomate())
      napo.agregarIngrediente(new PuñadoDeAceitunasVerdes())
      retornar napo
```

Y nos ayuda a tener en un método lo que sería la definición de una napo. De esta manera gano eso, pero no tengo una manera explícita de asegurar que las pizzas consistentes sean las que tengan masa y al menos un ingrediente. Podría combinar esto con los constructores de antes por ejemplo, y sigue creciendo la interfaz de Pizza.

De golpe caemos en que:

- El comportamiento de una pizza era muy muy simplecito, y estamos haciendo crecer y crecer su interfaz sólo para encargarnos de la creación, que no tiene que ver con cómo se comporta una pizza sino con cómo se conforma una pizza
- Hay reglas que dicen qué es una pizza consistente dependiendo de qué gusto sea, y al ser regla de negocio estaría bueno que esté modelado en nuestra solución y que no sea el usuario el que retiene esa info en la cabeza
- Estaría bueno tener una manera flexible de agregar nuevas pizzas que se construyan diferente sin tener que cambiar la interfaz de Pizza

Con estas cosas en la cabeza, podemos tratar de buscar más cohesión en esta solución y hacer que haya un objeto aparte que se encargue de la construcción de la Pizza, sin que tenga que ser la Pizza en sí.

Me gustaría poder enviarle a un objeto los siguientes mensajes:

agregarMasa
agregarQueso
agregarSalsa
agregarIngredientePrincipal
agregarAceitunas

Y que se encargue de armar la pizza como corresponda. Si la pizza es una napo, le pondrá las partes correspondientes a una napo. Si la pizza es fugazzeta, lo que corresponda también.

Al objeto que reciba esos mensajes lo vamos a llamar builder, y al que los envíe director:

```
clase MaestroPizzero
  metodo nuevaPizza
      builder.agregarMasa()
      builder.agregarQueso()
      builder.agregarSalsa()
      builder.agregarIngredientePrincipal()
      builder.agregarAceitunas()
      retornar builder.build()

clase PizzaBuilder
   constructor
      pizza = new Pizza()
   metodo agregarQueso
      pizza.agregarIngrediente(new Muzzarella())

clase NapoBuilder
  metodo agregarMasa
     pizza.masa(new ALaPieda())
```

```
clase FugazzetaBuilder
   metodo agregarMasa
      pizza.masa(new AlMolde())

clase NapoBuilder
   metodo agregarSalsa
      pizza.agregarIngrediente(new SalsaDeTomate())

clase FugazzetaBuilder
  metodo agregarSalsa
    // nada

... etc
```

Para refactorizarlo podemos hacer Factory Methods.

3.2 Abstract Factory

Imaginemos que en el sitio web de Starbucks se ofrece la posibilidad de armar una bebida personalizada a partir de una prearmada. El usuario arma su bebida preferida, combinando distintos ingredientes (base de café o crema, jarabes, toppings) y cuando está satisfecho con su bebida ideal, le pone un nombre y la publica en el catálogo de bebidas personalizadas de Starbucks. En el catálogo, la bebida dice cuánto sale y cuántas calorías tiene, y muestra cuántas personas la marcaron como favorita.
El precio de la bebida es $10 + la suma de los precios adicionales de sus ingredientes adicionales.
Las calorías son la suma de las calorías de sus ingredientes adicionales + las calorías del ingrediente base + las calorías de la leche.

Ahora bien, los de Starbucks dicen que tu bebida puede ser super personalizable, pero no te dejan mezclar cualquier cosa con cualquier cosa:

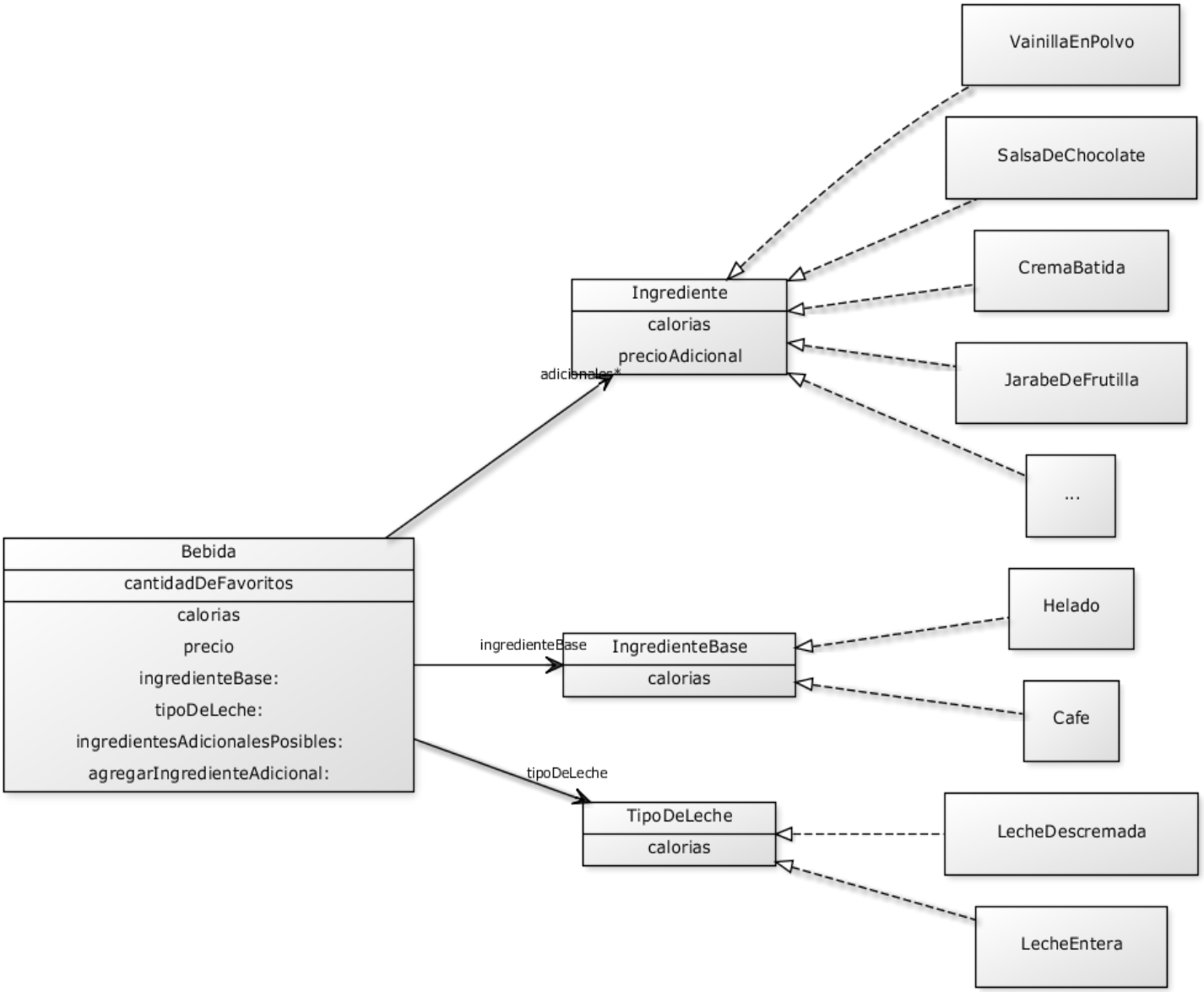
- Los frappuccinos en base a helado se hacen con leche entera y pueden tener jarabes de sabor frutilla, vainilla y/o chocolate, y como toppings pueden tener crema batida y/o salsa de chocolate o frutilla.
- Los frappuccinos en base a café se hacen con leche descremada y pueden tener jarabes sabor dulce de leche y/o chocolate, y como toppings pueden tener crema batida, salsa de chocolate o cacao en polvo.
- Los lattes son en base a café, se hacen con leche descremada y pueden tener jarabes sabor caramelo, vainilla y/o chocolate, y como toppings pueden tener crema batida, cacao en polvo, vainilla en polvo o canela.

El usuario debe poder arrancar su personalización partiendo de una de tres opciones base:

- Un frappuccino en base a helado
- Un frappuccino en base a café
- Un latte

A partir de eso, en la pantalla le tienen que aparecer los ingredientes adicionales que puede agregar sobre la bebida elegida. Una vez que el usuario elige todos los ingredientes adicionales que quiere, se le agregan a la bebida y está lista para ser publicada.

Se pide analizar la siguiente solución, proponer una solución alternativa y explicar ventajas y desventajas.




```
clase Bebida
  metodo de clase nuevoFrappuccinoDeHelado
      bebida = new Bebida()
      bebida.ingredienteBase(new Helado())
      bebida.tipoDeLeche(new LecheEntera())
      bebida.ingredientesAdicionalesPosibles([
            new JarabeFrutilla(), new JarabeVainilla(), new JarabeChocolate(),
            new CremaBatida(), new SalsaDeChocolate(), new SalsaDeFrutilla()
      ])
      retornar bebida

clase Bebida
  metodo de clase nuevoFrappuccinoDeCafe
      bebida = new Bebida()
      bebida.ingredienteBase(new Cafe())
      bebida.tipoDeLeche(new LecheDescremada())
      bebida.ingredientesAdicionalesPosibles([
            new JarabeDDL(), new JarabeChocolate (), new CremaBatida(),
```

```
            new SalsaDeChocolate(), new CacaoEnPolvo()
      ])
      retornar bebida


clase Bebida
  metodo de clase nuevoLatte
      bebida = new Bebida()
      bebida.ingredienteBase(new Cafe())
      bebida.tipoDeLeche(new LecheDescremada())
      bebida.ingredientesAdicionalesPosibles([
            new JarabeCaramelo(), new JarabeVainilla(), new JarabeChocolate(),
            new CremaBatida(), new VainillaEnPolvo(), new CacaoEnPolvo()
      ]
      retornar bebida
```

3.2.1 Solución con Abstract Factory

Podríamos pensar en una solución en donde agrupemos a las distintas partes de un café según “su familia”. Tendríamos a la familia de ingredientes de los frappuccinos de crema, a la familia de ingredientes de los frappuccinos de café y a la familia de los latte. A cada familia de ingredientes podríamos tenerlos en una “Fábrica” de cada bebida, que nos diera el objeto que corresponda cuando queramos agregar la base de nuestra bebida, su leche, y saber qué ingredientes adicionales hay disponibles.

De cada fábrica entonces nos interesa que nos diga:

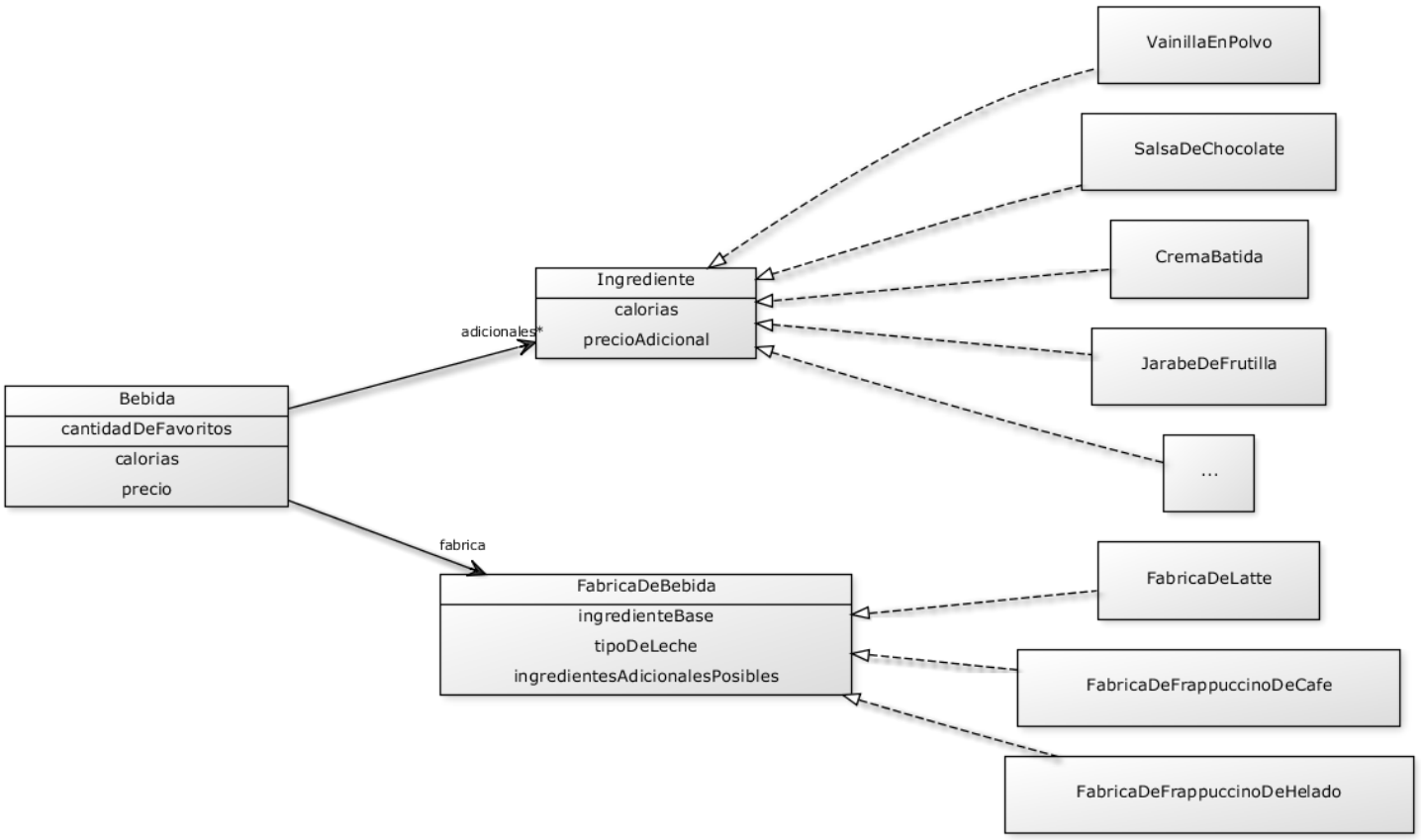
- El ingrediente base de la bebida
- El tipo de leche
- Los ingredientes adicionales posibles

Podríamos tener entonces 3 clases que respeten esa interfaz:

```
clase FabricaDeFrappuccinosDeHelado
  metodo ingredienteBase
      retornar new Helado()
  metodo tipoDeLeche
      retornar new LecheEntera()
  metodo ingredientesAdicionalesPosibles
      retornar [
            new JarabeFrutilla(), new JarabeVainilla(), new JarabeChocolate(),
            new CremaBatida(), new SalsaDeChocolate(), new SalsaDeFrutilla()
      ]

clase FabricaDeFrappuccinosDeCafe
  metodo ingredienteBase
      retornar new Cafe()

  metodo tipoDeLeche
      retornar new LecheDescremada()
```

```
  metodo ingredientesAdicionalesPosibles
      retornar [
            new JarabeDDL(), new JarabeChocolate (), new CremaBatida(),
            new SalsaDeChocolate(), new CacaoEnPolvo()
      ]
```

```
clase FabricaDeLattes
  metodo ingredienteBase
      retornar new Cafe()

  metodo tipoDeLeche
      retornar LecheDescremada()

  metodo  ingredientesAdicionalesPosibles
      retornar [
            new JarabeCaramelo(), new JarabeVainilla(), new JarabeChocolate(),
            new CremaBatida(), new VainillaEnPolvo(), new CacaoEnPolvo()
      ]
```

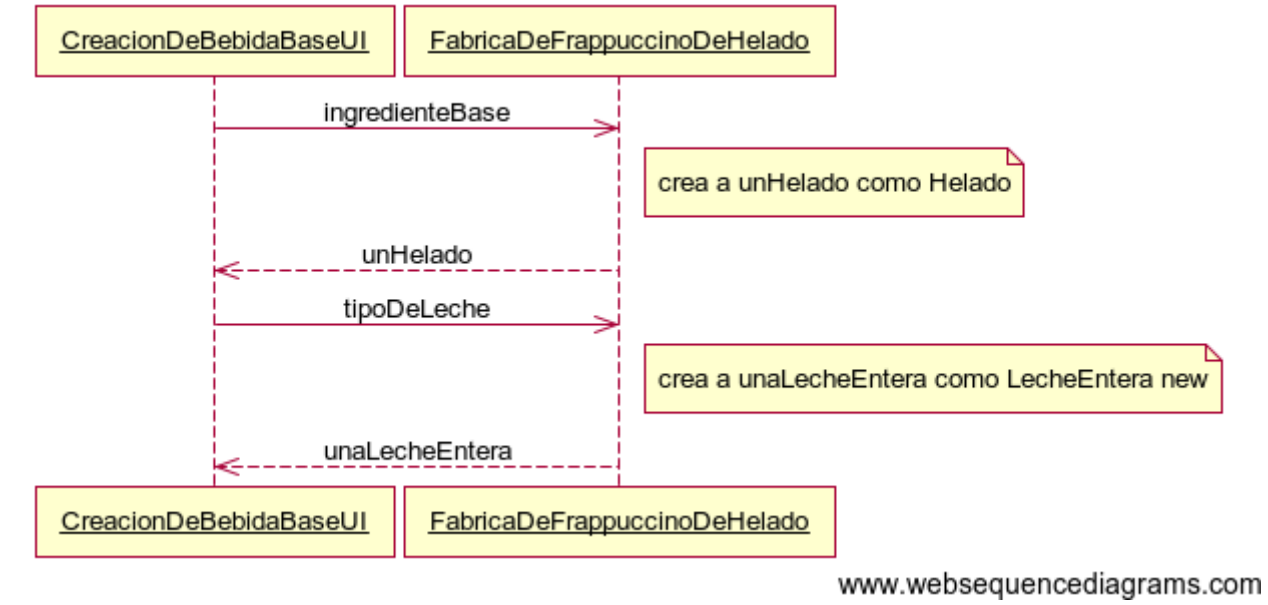


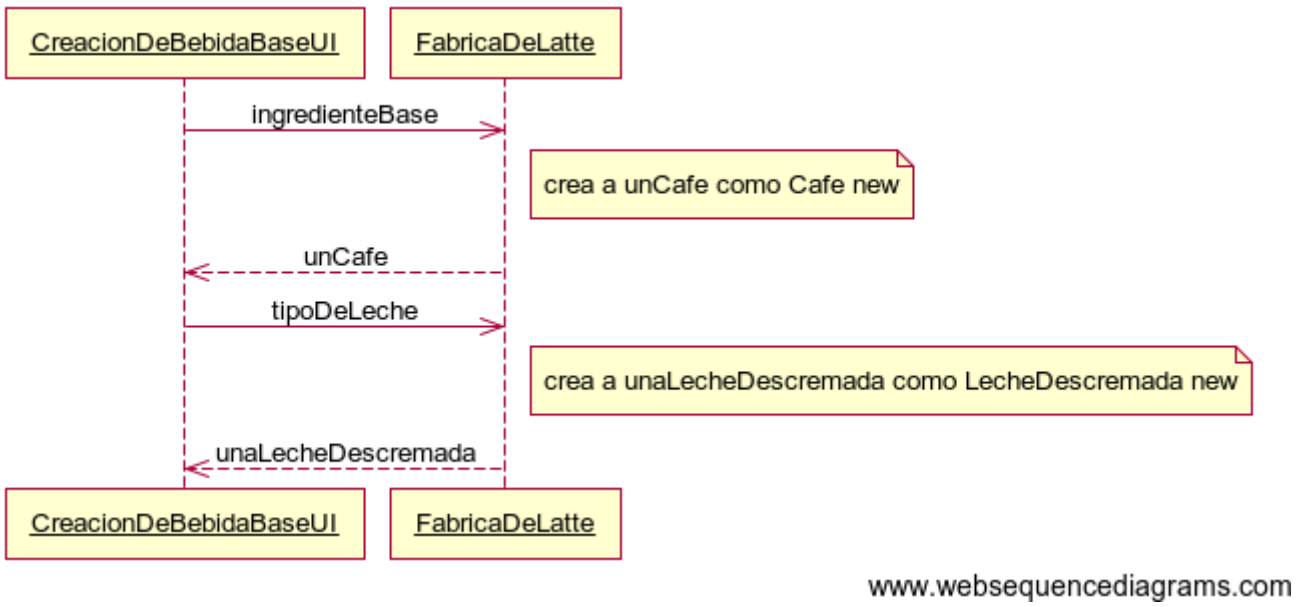
El cliente de estas fábricas puede despegarse de saber qué tipo de bebida le está ofreciendo al usuario construir, ya que puede tratarlas polimórficamente al pedirle cada una de las partes que tiene que conformar una bebida.

Se podría pensar que el componente UI que se encargue de mostrar 3 opciones de bebida base para

empezar, donde cada botón se asocia con una de las fábricas, y al elegir una opción recibiría el mensaje:

```
clase CreacionDeBebidaBaseUI
  metodo crearBebidaCon(unaFabricaDeBebida)
      this.fabrica = unaFabricaDeBebida
      bebida = new Bebida()
      bebida.setIngredienteBase(unaFabricaDeBebida.ingredienteBase())
      bebida.setTipoDeLeche(unaFabricaDeBebida.tipoDeLeche())
```





En ese paso se crearía la bebida “base”, y para un paso siguiente se podría tener otro componente de UI que esté asociado a la fábrica elegida, para ofrecerle al usuario los ingredientes adicionales posibles. La UI también se lo pediría a la fábrica con la que ya está asociada:

```
clase IngredientesAdicionalesUI
  metodo ingredientesAdicionalesPosibles
      retornar fabrica.ingredientesAdicionalesPosibles()
```

Esta solución se para sobre un patrón llamado **“Abstract Factory”**, que define una interfaz que ofrece objetos de una misma familia, que deberían tener que ver entre sí. En este caso se ve reflejado en la interfaz FabricaDeBebida, que dice que una “familia” de objetos que componen una bebida está formada por un ingrediente base, un tipo de leche y varios ingredientes adicionales, y cada clase concreta que implementa esa interfaz dice qué ingrediente base, qué tipo de leche y qué ingredientes adicionales son los que tienen que ver entre sí y pueden combinarse.

Ventajas:

- Deja bien explícito cuáles son los objetos que pueden usarse juntos de manera consistente
- Favorece a la extensibilidad a la hora de tener que agregar nuevas familias de productos
- Aísla la construcción de un nuevo objeto de saber cuáles tienen que ser las clases de sus componentes

Desventajas:

- Si se agrega un “nuevo miembro a una familia”, eso puede extender la interfaz y habría que cambiar cada fábrica concreta

//comentar que también se puede meter un builder

3.3 Prototype

Por otro lado, hay otro requerimiento: poder tomar como base para una bebida personalizada a otra que ya esté publicada en el catálogo, agregarle más ingredientes y publicarla como una bebida nueva.

//solución con prototype
Para resolver esto estaría bueno poder copiarse la estructura de una bebida ya creada y a partir de esa tener una nueva a la que le podamos agregar más ingredientes (consistentemente!).

Las bebidas podrían entender el mensaje “copy”, en donde se cree una nueva bebida igual a la receptora del mensaje:

```
clase Bebida
  metodo copy()
    //Se copia todo MENOS la cantidad de favoritos,
    // que debería ser nueva para la bebida nueva”
      retornar new Bebida(
            this.ingredienteBase,
            this.tipoDeLeche,
            this.ingredientesAdicionales.copy())
```

La cantidad de favoritos podría setearse en 0 al inicializar:

```
clase Bebida
  constructor
    cantidadDeFavoritos = 0
    ingredientesAdicionales = []
```

¿Y por qué en el último mensaje se le envía copy a los ingredientes adicionales?
Porque si la bebida nueva apunta a la misma colección de ingredientes adicionales que la bebida

“prototípica”, cuando agregue ingredientes en la bebida nueva también se agregarían en la bebida original! Entonces al copiar objetos, hay que tener mucho cuidado con cómo tratar a los objetos apuntados desde la copia.

//hacer diagrama de objetos

En Smalltalk por default todos los objetos entienden copy, y la implementación default es “shallow”, es decir, el objeto que recibe el mensaje copy se copia, pero los objetos a los que apuntan sus referencias no, las referencias de la copia van a apuntar a los mismos objetos a los que apunta el original. En Java en vez de copy es clone().

El patrón de diseño que se refleja en esta solución se llama **“Prototype”**, y se trata de crear nuevos objetos a partir de prototipos ya armados (instancias ya existentes).

Ventajas:

- Está bueno para crear fácilmente objetos que resultaron complejos de crear en un principio, como resultantes de un Decorator, Composite, o definidos por el usuario (que no tienen un proceso explícito en código que los pueda reproducir)

Desventajas:

- Implementar el mensaje copy a veces puede ser complicado en grafos complejos o con referencias circulares

Falta pensar en un detalle más: Los ingredientes adicionales que le pueda agregar a la nueva bebida, tienen que ser consistentes con los ingredientes que ya tenga la bebida! Lo que nos daba esa consistencia era el Abstract Factory, por lo tanto estaría bueno que las bebidas conozcan su fábrica para poder modificarlas consistentemente.

Modificamos:

```
clase CreacionDeBebidaBaseUI
  metodo crearBebidaCon(unaFabricaDeBebida)
      ~~fabrica = unaFabricaDeBebida.~~
      bebida = new Bebida()
      bebida.setFabrica(unaFabricaDeBebida)
      bebida.setIngredienteBase(unaFabricaDeBebida.ingredienteBase())
      bebida.setTipoDeLeche(unaFabricaDeBebida.tipoDeLeche())
```

Para cuando copiemos una bebida ya existente y usemos el IngredientesAdicionalesUI para agregar sus ingredientes, podemos indicarle cuál es la bebida base con la que tiene que trabajar, que sería la copia recién hecha.

```
clase IngredientesAdicionalesUI
  metodo ingredientesAdicionalesPosibles()
      retornar bebida.getFabrica().ingredientesAdicionalesPosibles()
```

3.4 Factory

Factory es una de las palabras más usadas y más imprecisas utilizadas en lo que respecta a diseño. Algunos usan el término "patrón factory" para referirse al Factory-Method, algunos para referirse al Abstract Factory,

otros para referirse a ambos y algunos para referirse a cualquier pieza de código que cree objetos.

Nuestra falta de entendimiento común sobre el término "Factory" limita nuestra habilidad de saber cuando un diseño podría beneficiarse con uno.

Una posible definición de Factory es un objeto que implementa uno o más Creation-Methods.