Unidad 3: Patrones de Creación

Introducción y orientaciones para el estudio

En el desarrollo de este módulo abordaremos:

1. Concepto de patrones de creación.
2. Clasificación y pre-requisitos.
3. Singleton, Fáctory (variantes Method, y Simple), Builder.

Objetivos

Pretendemos que al finalizar de estudiar esta unidad se logre:

- Comprender qué es y cómo se aplica un patrón de creación.
- Comprender su conveniencia de uso o no de un patrón de esta categoría.
- Comprender los esfuerzos de implementación, ventajas y desventajas.

Aclaraciones previas al estudio

En este módulo, el alumno encontrará:

- Contenidos
- Conceptualizaciones centrales
- Ejemplos

Se debe *tener* presente que los contenidos presentados en el módulo no ahondan profundamente en el tema, sino que pretenden ser un recurso de introducción y motivación y para que, a través de la lectura del material y la bibliografía sugerida y con el desarrollo de las actividades propuestas se alcancen los objetivos planteados.

Cada módulo constituye una guía cuya finalidad es facilitar el aprendizaje.

Patrones Creacionales.

Singleton (Única instancia)

(Tiempo estimado: 3 hs)

Nombre del patrón: Como dijimos, la traducción más cercana sería la de **"solterón",** en otras palabras para decir que el objeto creado o a crear estará solo será **único** (en el espacio de memoria del sistema, en tiempo de ejecución).

Clasificación del patrón: **Creacional.**

Motivación: En algunas ocasiones es muy importante poder garantizar que solo exista una instancia de una clase. Ocurre **cuando tenemos un recurso crítico y debemos administrar, coordinar o restringir su uso.**

Imaginamos la situación de varias impresoras disponibles cuando solo existe un solo pool (o cola de impresión) que maneja la impresión en la misma.

En este escenario es necesario asegurar que solo exista un objeto gestor de esa cola de impresión.

Basándonos en el ejemplo, necesitamos un patrón que nos permita controlar o restringir las situaciones que exigen la creación y el control de acceso a la instancia privilegiada para la tarea.

Esto es muy común en sistemas que tengan funciones concurrentes (que convergen a un mismo lugar.

Intención: Su intención consiste en **garantizar que una clase solo tenga una instancia** de una determina clase y que **proporcione también un único punto de acceso global a ella** (una vez que ya ha sido creada).

Aplicabilidad: Similarmente a la mención de la motivación, otras situaciones habituales de aplicación de este patrón suelen ser la **administración de un recurso físico o lógico** como puede ser el control de un mouse, o un archivo abierto en modo exclusivo, o cuando cierto tipo de dato que debe estar disponible para todos los demás objetos u entidades de la aplicación lo puedan acceder.

Segundo Nombre: No se le conoce.

Estructura: No tiene una estructura determinada, ya que se trata de una pequeña alternación en el código de la clase y en la forma en la que se accede a ella.

Participantes: Solo la clase que crea el objeto.

Colaboraciones: A primera vista no tiene, ya que todo lo que sucede, sucede dentro de la clase que crea el objeto. Sin embargo por los cambios que se introducen dentro de esa clase, la relación o colaboración del resto del sistema con esa clase cambia mucho obligándonos a una colaboración distinta (entre el sistema y como va a llamar y usar a esta clase singleton).

Consecuencias: A pesar de que el patrón muy sencillo de entender, y de usar, es uno de los que presenta bastantes consecuencias (que se podrían catalogar como negativas) que si no se saber de ellas o no se toman recaudos para manejarlos podría traer uno cuanto problemas:

- El objeto instanciado en memoria, no se irá nunca de ella, dado que la implementación tal como fue pensada inicialmente, no tiene medios para borrarlo. Deberé reiniciar el sistema para que se pueda borrar y liberar su espacio. Sin conocimiento o sin un uso controlado, es que algunos sistemas deben reiniciarse periódicamente (entre otras razones).
- Es costoso de introducirlo, no por la cantidad de líneas de código que implica escribir en la clase que se ocupa de manejar el recurso que tenemos que controlar, sino porque hay que cambiar (o usar otra forma de llamar) todo el sistema para que pueda usar a esa clase.
- La forma en la que se usa la clase no es habitual (dado que para crear un objeto "singleteado", ya no habrá que usar más la palabra "new").
- Una vez introducido en el sistema, su implementación se pierde dentro del sistema. No es fácil de verlo o reconocerlo en un diagrama, ya que el cambio necesario para implementarlo es simplemente unas pocas líneas de código. Para mitigar o reducir lo anterior se sugiere agregarle la palabra "singleton" adelante del nombre de la clase, para poderlo ver con claridad cuando se busca dentro del código (e.g: "SingletónColaDeImpresión").
- Puede traer complicaciones en sistema multi hilo (mult-thread), generado posibles bloqueos o loqueos dentro del sistema (dead-locks entre los hilos).
- Puede traer problemas con su descendencia (si heredamos de un objeto base que ya es singleton), en tiempo de ejecución (ej.: si no se establecen controles, la aplicación podría crearme un objeto ya restringido cuando quizás no quisiera que esto suceda, o que se sea restringido por un objeto padre y no del hijo con el que estoy trabajando).
- Controles adicionales harán más complejo la implementación, y quizás esto tampoco sea deseable.

Implementación (Resumen): El patrón *singleton* provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea inestancable directamente.
- Al estar internamente auto-referenciada, en lenguajes como Java, el recolector de basura no actúa.
- < ver el Detalle de Implementación >

Código de ejemplo: < ver el Detalle de Implementación >

Usos conocidos: Además de los ya mencionados en la motivación y en la aplicabilidad, podría ser usado para conexiones de base de datos, colas de mensajería, control de sesiones, control de instancias activas de la aplicación, etc.

Patrones relacionados: Puede relacionarse casi con cualquier otro patrón. Como ejemplo podemos citar un patrón Fachada, un patrón Fáctory, un patrón Proxy, etc.

⇨ Detalle de Implementación

Por un momento, permitámonos un lenguaje más informal, solo a efectos de ser prácticos y de poder entender más claramente lo que el patrón pretende hacer, como lo logra, y para que nos sirve.

Recordemos que el patrón propone: **1) controlar que exista una sola instancia de un objeto**. Esto sería que pueda crearse, si no existe ninguno creado previamente, y **2) si existe algún objeto ya previamente creado, que se pase la referencie al mismo** para que pueda ser posible su uso.

Estas dos propuestas implican que alguien tiene que tener alguna lógica que permita estas dos cosas. Pero quién o qué?!

Tratando de mantener las cosas simples, y descartando tener que pensar en otro objeto que se ocupe[1], la cosa más simple que nos queda por hacer es **trabajar con la propia clase que crea el objeto** para que haga algún control de lo que está construyendo.

Trabajar con la clase que construye un objeto, implica que tenemos que trabajar con métodos y variables de clase. Es decir **trabajar con métodos y atributos "estáticos"** (se suelen llamar del mismo modo: "static").

Puede que hasta ahora, no se haya presentado o explicado en profundidad el tema, dado que quizás la necesidad no estaba tan presente.

Las variables estáticas, son aquellas en las que la propia clase guarda información que solo ella puede necesitar durante la construcción de sus objetos. **Son independientes de las que usa un objeto** instanciado por ella.

Los métodos de clase, son los que usan las variables de clase, y que también **son de uso exclusivo de las clases**.

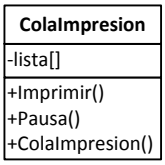
Volviendo a nuestro tema, queremos entonces que la clase nos ayude a construir **un solo objeto, lleve el control y me acceso a él** (en el caso que ya esté construido). Veámoslo con un ejemplo:

[1] **Nota**: Es necesario decir que existe una variante de este patrón que hace específicamente esto: crear un objeto que registra la lista de los objetos que deben ser "singleteados" o controlados en su cantidad de instancias. A esta variante se la llama "Singleton Registry". Es una variante que no veremos en esta materia.

El Objeto a controlar (o recurso crítico):

Lo primero que necesitamos tenés definido es el recurso crítico que queremos controlar. En este caso, para hacerlo fácil, pensamos en una clase cola de impresión, para la cual podemos ver que se necesita que exista solo una en nuestro sistema de ejemplo para poder centralizar la impresión de documentos. Debajo, la representación UML y su código:

Diagrama UML



Código Original

```
public class ColaImpresion {
  private lista; // un atributo (objeto)
  public ColaImpresion ( ) { … } // el constructor
  public Imprimir( )  { … }  // método de negocio
  public Pausa( ) { … }  // método de negocio
}
```

Hasta aquí solo presentamos el objeto que necesito controlar. Para el ejemplo no interesa la lógica de negocio.

Controlando y Devolviendo la creación:

Lo primero que podemos pensar es simplemente, agregar alguna variable estática que cuente cuantos objetos lleva creada la clase. De esa manera construyo el objeto si el contador está en cero y sino no construyo nada.

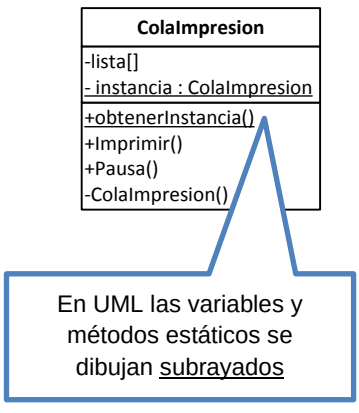
Bueno, esto funciona a medias, ya que el constructor debe construir siempre, y además si ya se construyó un objeto no tengo manera de que me lo pase.

La simple solución en 4 pasos será:

1. Tener un **atributo de clase/estático que se guarde la instancia** del objeto que se está creando y que se quiere controlar.
2. Como tengo un atributo estático, y el constructor no puede no construir, **deberé tener un método estático** para poder manejar el atributo.
3. Tener una condición que cuando el objeto **no haya sido nunca creado lo cree, y si ya fue creado lo devuelva**.
4. Para que todos usen el método estático, **deberé ocultar el constructor** para que nadie lo use.

De esta forma, la solución se ve con los sencillos siguientes cambios:

Diagrama UML



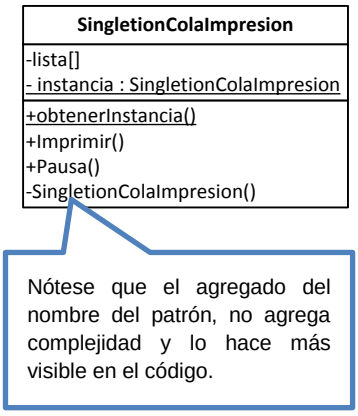
Código Implementando el Patrón

```
public class ColaImpresion {
  private static ColaImpresion instancia;          (1)
  private lista; // un atributo (objeto)
  public static ColaImpresion obtenerInstancia( ) {   (2)
  if ( instancia == null ) {
    instancia = new ColaImpresion()                (3)
    }
  return instancia;
  }
  private ColaImpresion ( ) { … } // el constructor   (4)
  public Imprimir( )  { … }  // método de negocio
  public Pausa( ) { … }  // método de negocio
                                                   }
```

Recomendación y Comentario:

⇨ Para remediar en alguna medida la consecuencia negativa de la poca visibilidad del patrón una vez implementado, se sugiere que se le agregue a la clase la palabra "singleton" delante del nombre de negocio para que podamos ubicar más rápidamente al patrón o saber cómo tenemos que usarlo. De este modo la implementación quedaría como se indica debajo. Esta recomendación será de utilidad para todos los patrones (siempre y cuando podamos nosotros manejar los nombres de las clases y objetos.

Diagrama UML



Código Mejorado

```
public class SingletonColaImpresion {
 private static SingletionColaImpresion instancia;          1
 private lista; // un atributo (objeto)
2 public static SingletonColaImpresion obtenerInstancia( ) {
  if ( instancia == null ) {
   instancia = new SingletionColaImpresion()          3
   }
  return instancia;
 }
4 private SingletionColaImpresion ( ) { … } // el constructor
  public Imprimir( ) { … } // método de negocio
  public Pausa( ) { … } // método de negocio
                                                    }
```

⇨ Nótese también que implementar un Singleton tiene una consecuencia de impacto en todo el sistema, ya que si antes creaba un objeto mediante la sentencia "new", ahora lo deberé hacer usando el nuevo método estático "obtenerInstancia". Se ilustra este cambio con un ejemplo de código:

Creación antes del Singleton

```
//creación del objeto
ColaImpresion ci = new ColaImpresion() ;

//uso del objeto
ci.Imprimir();
```

Creación después del Singleton

```
//creación del objeto
SingletonColaImpresion ci = SingletonColaImpresion.obtenerInstancia() ;

//uso del objeto
ci.Imprimir();
```

Aquí se ven algunas de las consecuencias más importantes del uso del patrón:

1) Tengo que cambiar en todos lados la forma en la que se instanciará el objeto.
2) Dejo de usar la palabra "new" teniendo que hacer referencia a la clase.
3) Aumenta la dependencia con la clase dado que tengo que saber y usar el nombre de la misma en todos lados (relación con el acomplamiento de clase).

Factory (Fabrica)

(Tiempo estimado: 3 hs)

Nombre del patrón: Se postuló originalmente como "Factory Method", (o método fábrica/construcción), luego en una versión simplificada como Simple Factory (o sencillamente Factory).

Clasificación del patrón: Creacional

Intención: Facilitar en general la creación de objetos de un tipo o familia (mediante un método o interfaz o clase). "Factory Method" o Simple Factory, tendrá la responsabilidad de instanciar determinados objetos de tipos similares o de distintos tipos mediante métodos de creación con una o varias determinada firmas.

Siembre con la intención de simplificar la lógica de los objetos de negocio (u otro tipo), delega la fabricación o instanciación de objetos producto en otra clase (especial para ese uso) o bien define una interface para la creación de objetos de forma normalizada o generalizada.

Motivación: Normalmente en una aplicación de alcance importante o un framework se nos presenta la necesidad de tener relaciones entre objetos y muchas veces de cierto nivel de abstracción. De este modo, la misma aplicación o framework deberán ser capaces de instanciar los distintos objetos producto concretos que correspondan de acuerdo a los distintos escenarios. …

Simple Factory y Factory Method ofrece una solución concreta: encapsular el conocimiento de creación de los objetos en clases y subclases separadas quitando esa tarea de los objetos de negocio.

Aplicabilidad:

- Cuando una clase (la clase que usará el método Fáctory o a la clase Factory Simple) puede no prever la clase de objetos que tiene que crear.
- Cuando una clase quiere que sus subclases decidan qué objetos deben crean.
- Cuando las clases delegan responsabilidades a una de entre varias subclases auxiliares, y queremos localizar en qué subclase concreta se ha delegado.

Segundo Nombre: Factory Simple. Más que un segundo nombre es una forma derivada simplificada de la originalmente postulada para el Factory Method.

Estructura: Para la versión Factory Method, nos referimos a la firma de un método que construye, que luego distintas clases y subclases se ocuparán de implementar.

Para la versión Factory Simple, se trata simplemente de una clase en la que se alojan los métodos que construyen los objetos productos que necesitamos.

Participantes: Tanto para la versión Fáctory Method, como para la versión Factory Simple, habrá una clase que aloje los métodos de construcción y una clase producto (que es el objeto específico a fabricar).

Nota: consideramos solo a los participantes y colaboraciones concretos.

Colaboraciones: Las principales son la de la clase factoría que construye, que crea a los objetos necesarios, con la de los productos (los objetos a ser fabricados).

Consecuencias:

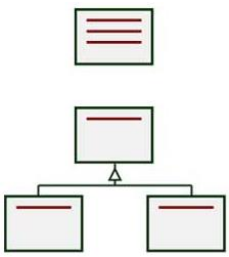
Positivas:

- Elimina la necesidad de enlazar por código a clases específicas de una aplicación que se deben construir.
- El cliente (o la aplicación), simplifica su código ya que solo tiene que pedir los objetos producto que necesita a la clase que implementa el Factory.
- La creación de objetos es más flexible. Se puede especificar (o trabajar con subclases) fácilmente.

Negativas:

- Algunos autores podrían decir que agrega más de complejidad, ya que implica desarrollar y mantener más objetos en el código y en memoria, sin embargo es tolerable esa inclusión y resulta una consecuencia negativa poco significativa respecto a las positivas.

Implementación (Resumen):



Simple Factory

Se trata de una clase con la responsabilidad de crear objetos de otras clases (productos). Es una clase concreta, y no delega en subclases, y sus métodos pueden ser también estáticos. Esta versión simplificada del patrón puede evolucionar a un Factory Method o un Abstract Fáctory.

Factory Method

Define una interfaz (método o métodos) para crear un objeto (producto), pero deja que sean las subclases quienes decidan qué clase se va a instanciar. Permite que una clase delegue sus subclases a la creación de objetos.

Código de ejemplo: <ver el Detalle de Implementación>

Usos conocidos: Ambos tipos de Factory se usan ampliamente en cualquier tipo de aplicación, no solamente en aplicaciones grandes o en frameworks.

Patrones relacionados: Mencionamos primero los citados en la documentación del Gof, tales como el Abstract Fáctory, el Template Method, y el Prototype. El uso combinado de estos patrones, no siempre es fácil de ver. No siempre es posible tener estos patrones mencionados en nuestra aplicación para poderlos combinar.

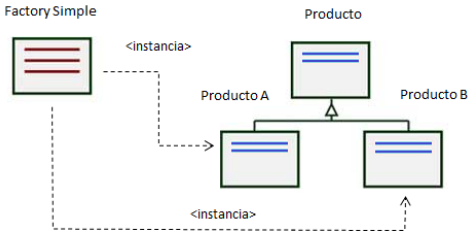
Sin embargo, el uso tan generalizado del Factory, nos lleva a relacionar y/o combinar el patrón con patrones tales como el: Facade (cuando se combina una fachada con un Fáctory), con objetos de manejo de colecciones en memoria (Managers, u objetos que se ocupan de Cacheo), hasta con otros patrones de manejo de persistencia (dependiendo de la estrategia que se elija usar).

⇨ Detalle de Implementación

Factory Simple

El siguiente esquema muestra la idea general de la implementación:

Esquema Concreto



Comentarios

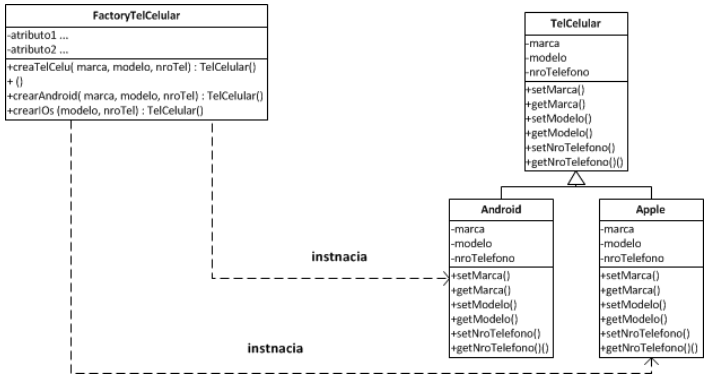
Se trata simplemente de una clase que tiene métodos para crear objetos producto de una familia (generalización).

La clase Fáctory puede tener un método con parámetros (y la inteligencia para para entender que valores tienen y crear el objetos que corresponda), o bien puede tener distintos métodos para crear directamente el objeto de cada tipo de la generalización que se necesite.

En este gráfico se ven los participantes y colaboraciones concretas más claramente.

La siguiente es una representación UML de una implementación (sin código) ya que para este patrón resultaría trivial:

Representación UML



Comentarios

La representación muestra una clase Factory Simple (llamada para el caso FacforyTelCelular), que simplemente provee métodos para crear objetos de la familia TelCelular.

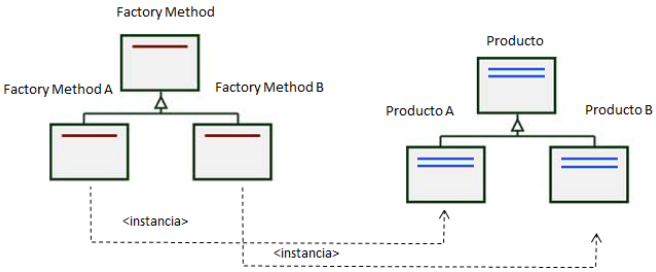
Dado que la familia TelCelular tiene un hijo concreto Android y otro Apple, la Factory Simple muestra varios métodos para poderlos crear.

El primero es un solo método con suficientes parámetros para poder resolver las dos creaciones, y dos métodos más específicos para tener creaciones distintas y hasta con parámetros distintos.

Factory Method

El esquema muestra la idea general:

Esquema Concreto



Comentarios

Se trata de tener el método de construcción en una clase abstracta, dando la posibilidad de tener otras clases hijas que implementen este método en forma concreta y que se sean estas clases las que realmente construyan los objetos producto que son necesarios.

Esta versión, si bien puede parecer más compleja es más flexible en términos de mantenibilidad ya que agregar un producto nuevo con su clase nueva de construcción permite extender la solución, no modificar la existente y evitarnos gastar tiempo en pruebas y/o riesgos en romper algo que ya esté funcionando.

Builder (Constructor)

(Tiempo estimado: 3 hs)

Nombre del patrón: "Builder" o simplemente "constructor".

Clasificación del patrón: Es Creacional.

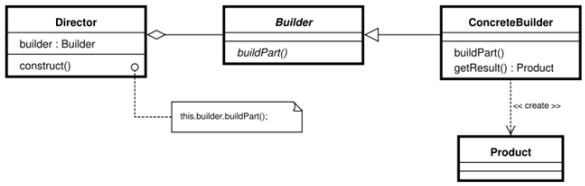
Motivación: Es usado cuando necesitamos la creación de una variedad de objetos complejos basados en un objeto principal al que llamamos producto. La necesidad se presenta cuando el producto se compone de una variedad de partes a las que hay que crearlas por separado, que se suelen crear en una determinada secuencia, y que suelen o deben agregarse o ensamblarse con ciertas condiciones.

Intención: Abstraer el proceso de creación de un objeto complejo, centralizando ese proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

Aplicabilidad: Similarmente a la mención de la motivación, otras situaciones habituales de aplicación de este patrón suelen. Cuando tenemos una complejidad tal para construir un objeto, para la cual se necesita un algoritmo para crearlo que deba ser independiente de las partes que componen y de cómo se ensamblan. Cuando el proceso de construcción deba permitir distintas representaciones para el objeto que se construye.

Segundo Nombre: No tiene.

Estructura:



Participantes: Tenemos al "Director" (que es el objeto/clase al que se le pide la construcción del producto). El "Builder", la abstracción que define el producto a crear. El Builder concreto, la implementación del Builder y el que construye y/o reúne las partes necesarias para la construcción del producto específicamente. Finalmente tenemos al producto, que la cosa que queremos construir.

Colaboraciones: El cliente o contexto que necesita usar el patrón, crea el objeto director y lo configura con el objeto constructor deseado. El director avisa al constructor cuando una parte del producto tiene que ser construida. El constructor gestiona las peticiones del director y añade partes al producto. <Ver detalles de la colaboración>

Consecuencias: Favorables: Desacopla al director del constructor, permitiendo nuevas representaciones de objetos cuando cambia el constructor. Separa el código de la construcción de la representación. Nos da un control preciso sobre el proceso de la construcción.

Desfavorables: entre el Director y el Builder, se generan dos estructuras más dentro del modelo que van a requerir un refactory (si no fueron pensadas desde el inicio). La implementación es compleja y requiere esfuerzo y es posible que impacte en el resto del sistema. Puede implicar una reestructuración mayor a nivel de componentes. El beneficio debe ser claro antes de implementarlo.

Usos conocidos: Este patrón es útil en alguno de los siguientes supuestos:

- Cuando la lógica o el algoritmo para crear un objeto complejo puede independizarse de las partes que componen el objeto y de cómo son ensambladas.
- Cuando el proceso de construcción debe permitir distintas representaciones para el objeto que se construye.
- Cuando el objeto a construir es complejo y sus distintas configuraciones son limitadas. En caso de que necesitemos un objeto complejo pero cada una de sus partes deba ser configurado de forma individual (en el ejemplo que nos ocupa, se trataría de definir cada elemento "al gusto" del consumidor en lugar de objetos predefinidos), este patrón no será una buena idea, ya que será necesario realizar el proceso de asignación de cada elemento paso a paso.

Patrones relacionados: En ocasiones se lo relaciona al patrón Abstract Factory es similar al patrón Builder debido a que también puede construir objetos complejos. Pero el Builder es más específico, ya que se ocupa de partes, de secuencia y de ensamblar esas partes en el producto. La principal diferencia es que el patrón Builder se centra en construir un objeto paso por paso, patrón Abstract Factory pone énfasis en familias de objetos productos (simples o complejos). El patrón Builder retorna el producto como un paso final, pero el patrón Abstract Factory lo devuelve inmediatamente. Un objeto compuesto (o un objeto "Composite" (que es un patrón que veremos en la cursada)) es lo que el constructor a menudo construye.

⇨ Implementación y ejemplo de código

El siguiente ejemplo nos muestra un diagrama de clases, en el que se representa al código de ejemplo adjunto.

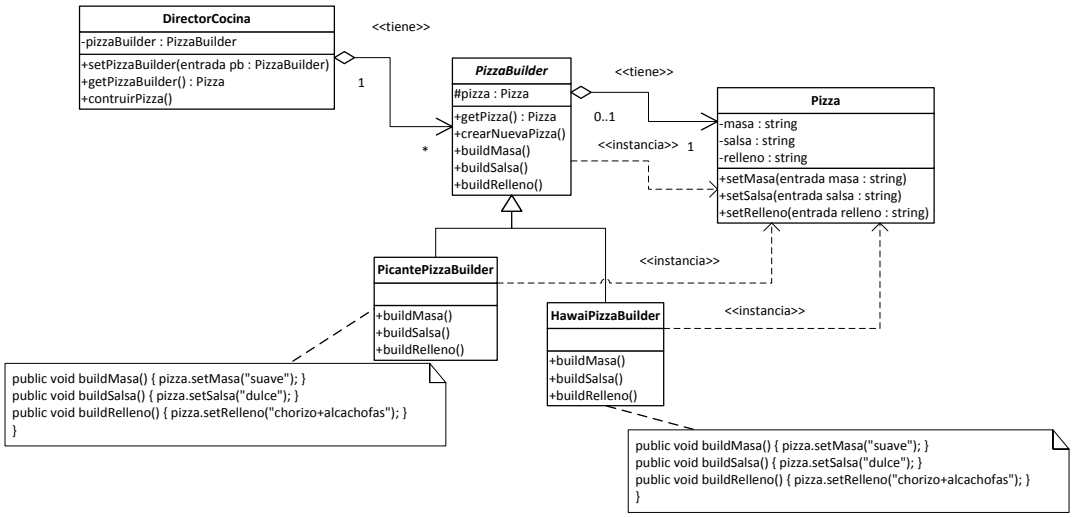
Codigo Ejemplo Builder v02.txt

Se trata de un objeto director que se llama “cocina”, quien puede contener y manejar a cualquier constructor de pizzas que le configuren.

Luego tenemos a la abstracción de constructores de pizzas (el PizzaBuilder), a los constructores concretos de pizzas (el constructor Picante Pizza, y Hawái Pizza), y a la pizza, que es lo que finalmente todos los constructores saben construir.

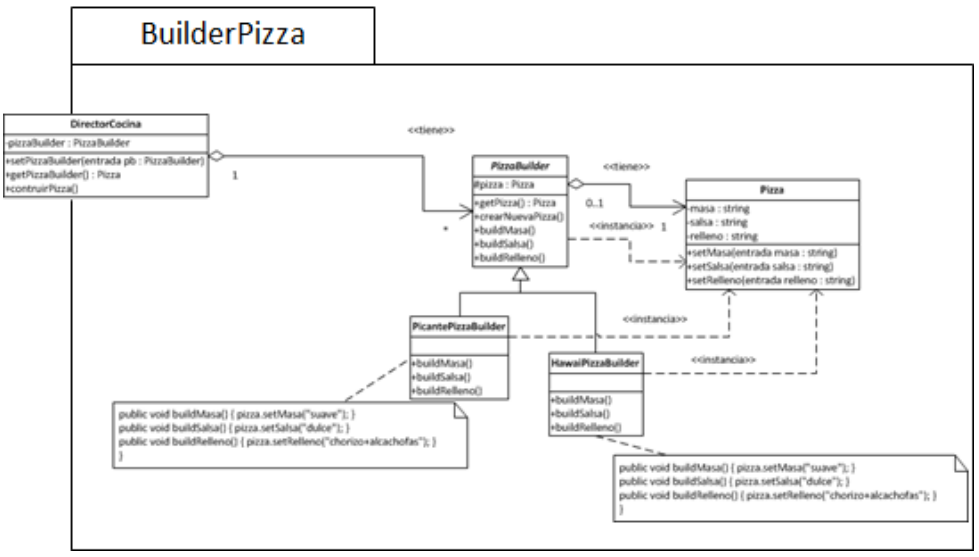
En este caso la construcción es simple, solo se construye al producto (Pizza), y no hay partes de la misma que se tengan que crear por separado que el constructor tenga que ensamblar.

La idea es que todo mundo (contexto o clase cliente) que necesite una pizza se la pida al director cocina, y ésta le simplifique la creación.



Este patrón por su esfuerzo de implementación, la relación de las clases que produce, su complejidad y su reusabilidad, se suele implementar en un paquete separado, estableciendo case un módulo de reutilización completo.

En la imagen de abajo, se ve como se suele operar con el mismo y como el Director hace las veces de interlocutor con el contexto, clase cliente, o resto del sistema que lo reutiliza.



⇨ Detalles de la colaboración

El siguiente diagrama de secuencia nos muestra detalles de la colaboración, y también algunas opciones de implementación.

Para el caso, el cliente (contexto o clase que utiliza el patrón), crea primero el Builder, para luego pasarlo como argumento para configurar el director y obtener el producto.

Luego se le pide al director la construcción del producto, y éste ejecutará las tareas, en el orden, la secuencia o ciclo necesario y con las condiciones que correspondan.

Con todo esto, creará el producto. A partir de ahí e cliente se lo podrá pedir al Builder, o a Director (en un segundo momento), o el Director podrá devolverlo de inmediato si puede. <el cuadro azul indica las opciones>.