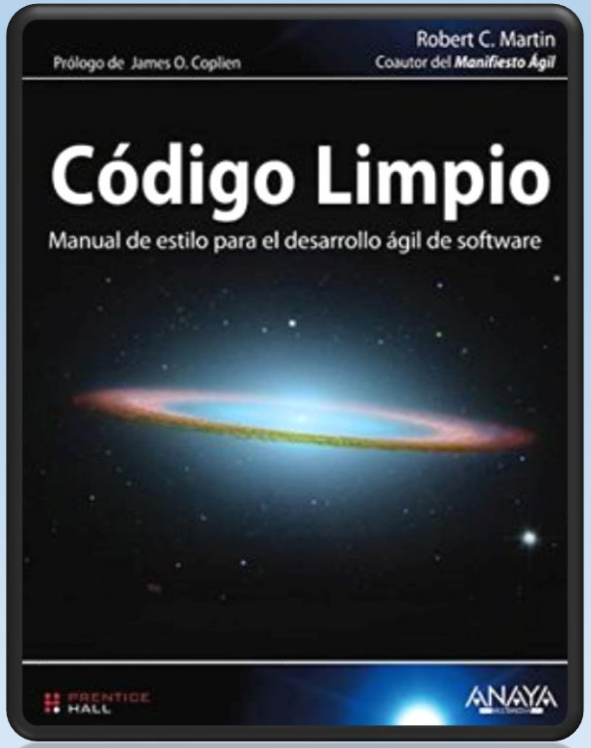
Código Limpio

Manual de estilo para el desarrollo ágil de software



Capitulo 11: Sistemas

Alumno: Octavio Lucardi Fierro

Curso: 6º11º

“La complejidad es letal. Acaba con los desarrolladores y dificulta la planificación, generación y pruebas de los productos”

Ray Ozzie, CTO, Microsoft Corporation

Separar la construcción de un sistema de su uso:

La construcción es un proceso **muy diferente** al uso. Los sistemas de software deben separar el proceso de inicio, en el cual se crean los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que toma el testigo tras el inicio.

El proceso de inicio es un aspecto que toda aplicación debe abordar. Desafortunadamente, muchas aplicaciones no lo hacen. Por ello existen técnicas para llevar esto a cabo:

Construcción del sistema

Uso del sistema

Técnica de la inicialización / evaluación tardía

No incurrimos (caemos en la falta) en la sobrecarga de la construcción a menos que usemos el objeto realmente, como resultado el tiempo de inicio se puede acelerar y evitamos que se devuelva “null”.

Pero si tenemos Dependencias (del objeto) no podremos compilar la aplicación sin resolver esto primero.

Separar Main

Una forma de separar la construcción, consiste en trasladar todos los aspectos de la construcción a Main o a módulos individuales por main, y diseñar el resto del sistema suponiendo que todos los objetos se han creado y conectado correctamente.

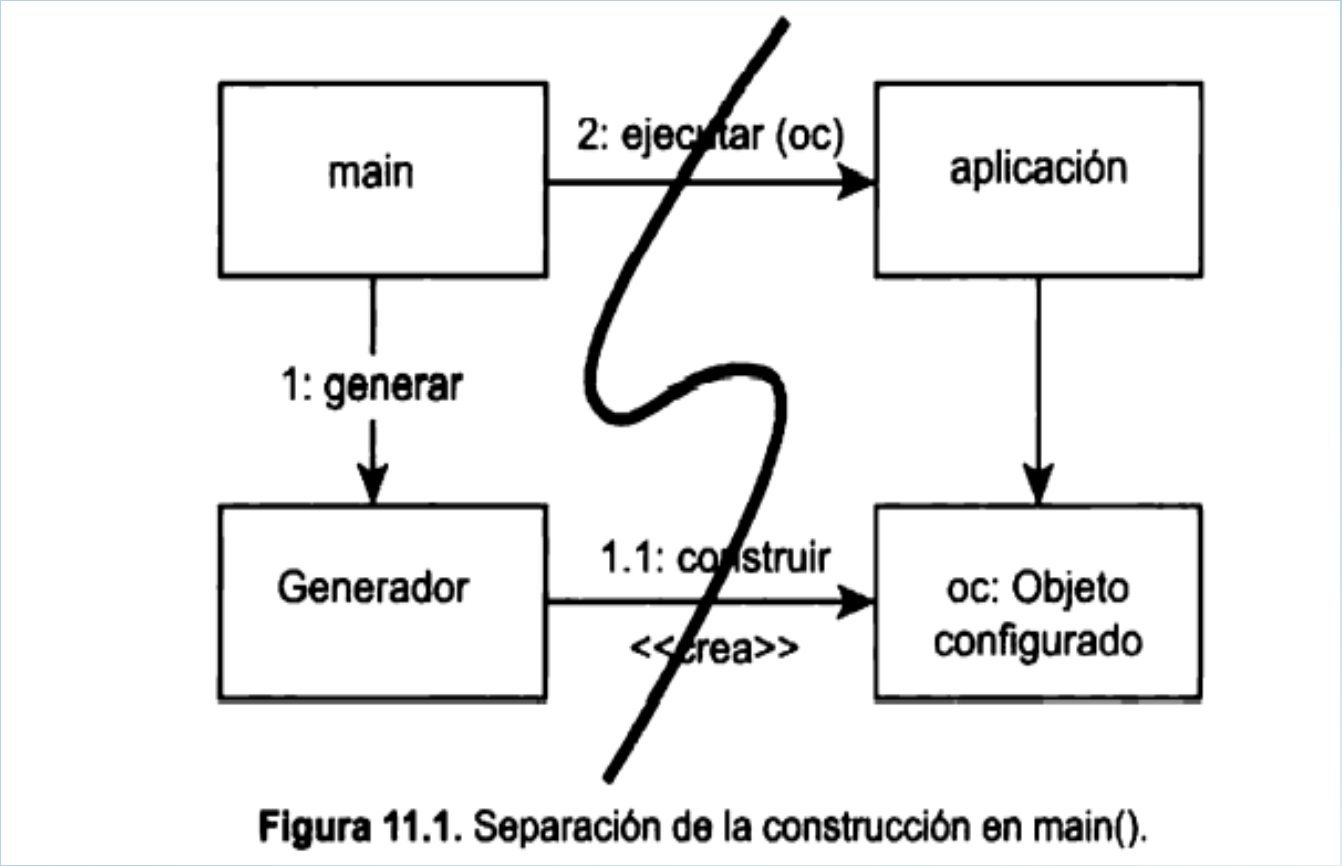
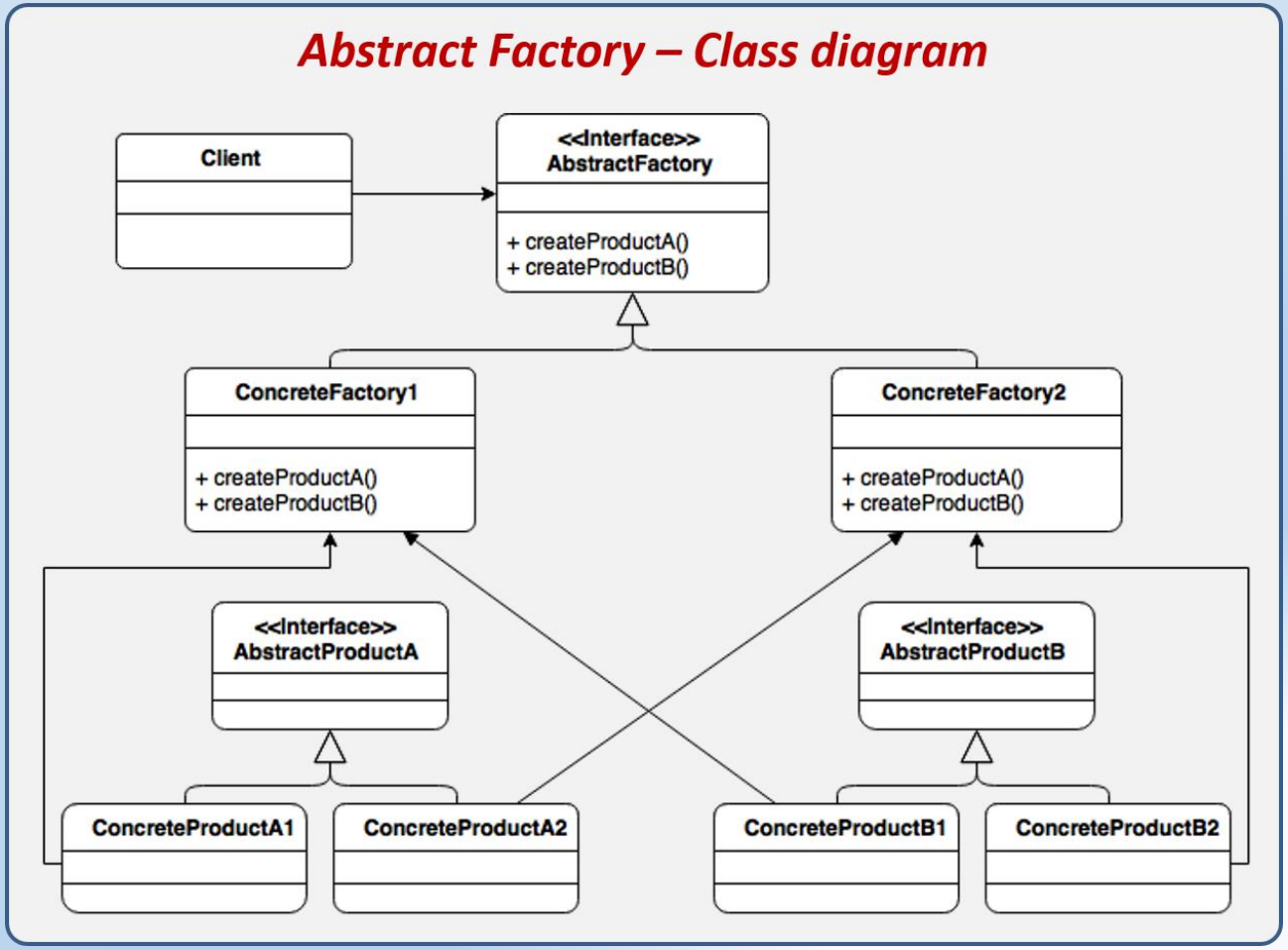


Figura 11.1. Separación de la construcción en main().

El flujo de control es fácil de seguir. La función main crea los objetos necesarios para el sistema, los pasa a la aplicación y esta los utiliza, todas las flechas van en la misma dirección, alejándose de main, lo que significa que la aplicación no tiene conocimiento de main ni del resto del proceso de construcción. Simplemente espera a que todo se haya construido correctamente.

Factorías

En ocasiones la aplicación será responsable de crear un objeto. En este caso podemos usar el **patrón de factoría abstracta (Patrón de diseño)** para que la aplicación controle cuando crearlo, pero manteniendo los detalles de dicha construcción separados del código de la aplicación.



Inyectar Dependencias

Un potente mecanismo para separar la construcción del uso es *la Inyección de dependencias*, la aplicación de **Inversión de control** (un método de trabajo en el cual el flujo de ejecución de un programa se invierte) a la administración de dependencias. La inversión de control pasa responsabilidades secundarias de un objeto a otros dedicados a ese cometido. Aparte de esto la inyección de dependencias va un paso más allá. En un programa la clase no hace nada directamente para resolver sus dependencias, por el contrario ofrece **métodos de establecimiento o argumentos de constructor** (o las 2) que se usan para inyectar dependencias.

Mas sobre las dependencias

En el proceso de construcción el contenedor de inyección crea instancias en los objetos necesarios (lo hace bajo demanda) y usa lo **anteriormente mencionado** para conectar las dependencias. Por su parte objetos dependientes suelen especificarse a través de un archivo de configuración o mediante programación en un módulo de construcción de propósito especial.

Ejemplo: La estructura Spring (es un framework de código abierto para la creación de aplicaciones empresariales Java) proporciona el contenedor de inyección más conocido de Java, los objetos que se van a conectar se definen en un archivo de configuración XML.

Evolucionar

Conseguir sistemas perfectos a la primera es un mito. Por el contrario, debemos **implementar hoy**, y re factorizar y **ampliar mañana**. Es la esencia de la agilidad iterativa e incremental. El desarrollo controlado por pruebas, la refactorización y el código limpio que generan hace que funcione a nivel de código.

¿Pero qué sucede en el nivel del sistema? ¿La arquitectura del sistema no requiere una planificación previa? Sin duda no puede aumentar incrementalmente algo sencillo a algo complejo ¿O sí?

Proxies de Java

Un proxy recibe solicitudes del cliente, realiza parte del trabajo (control de acceso, almacenamiento en caché, etc.). Los proxies de Java son útiles en casos sencillos, como envolver invocaciones de métodos en objetos o clases concretas, sin embargo, los proxies dinámicos proporcionados en el JDK (Java Development Kit) solo funcionan en interfaces. Para aplicarlos a clases se debe usar una biblioteca de manipulación de código de bytes (CGLIB (Code Generation Library) o Javassist).

Aspectos de AspectJ:

Por último la herramienta más compleja de separación a través de aspectos es el lenguaje AspectJ, una extensión de Java que ofrece compatibilidad de primer nivel para aspectos como construcción de modularidad. Ofrece un conjunto de herramientas avanzadas para la separación de aspectos.

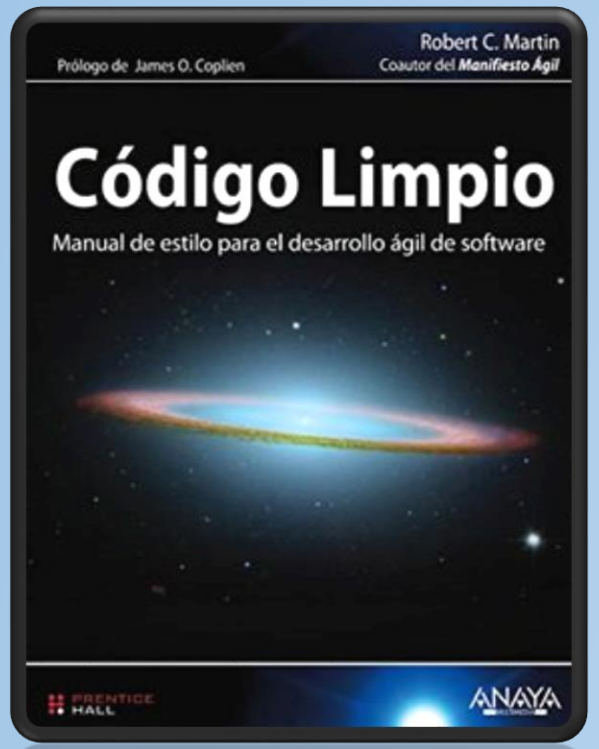
El inconveniente de este lenguaje es la necesidad de adoptar nuevas herramientas y aprender nuevas construcciones del lenguaje. Igualmente estos problemas se han mitigado gracias a la introducción de un formato de anotación de AspectJ, donde se usan anotaciones de Java 5 para definir aspectos con código puro de java.

Optimizar la toma de decisiones

La modularidad y separación de aspectos da como resultado la **descentralización de la administración** y la toma de decisiones. En estos tipos de sistemas amplios, **no** debe haber **una** sola persona que adopte todas las decisiones. Sabemos que conviene delegar las responsabilidades en las personas más calificadas, solemos olvidar que también conviene posponer decisiones hasta el último momento, no es falta de responsabilidad, nos permite tomar decisiones con la mejor información posible.

Si decidimos bastante pronto tendremos menos información sobre nuestro cliente, reflexión mental sobre el proyecto y experiencia con las opciones de implementación.

Fin



Fuente principal:

- *Código Limpio: Manual de estilo para el desarrollo ágil de software*

Fuentes Secundarias:

- *https://openwebinars.net/blog/que-es-spring-framework/*
- *https://itblogsogeti.com/2015/10/29/inyeccion-de-dependencias-vs-inversion-de-control-eduard-moret-sogeti/*