Java Intermedio 2019 - Módulo III

November 27, 2019

1 Java Intermedio - Módulo III

1.0.1 Programa

Tipos genéricos y colecciones. Interfaces. List, Set, Queue, Map. Formas de iterar a través de colecciones. Patrones de diseño. Utilización de librerías externas. Introducción a Maven, Hibernate y Java Persistence API (JPA).

1.1 Interfaces

Una interfaz es como una clase Java pero solo tiene constantes estáticas y métodos abstractos. Java usa la interfaz para implementar herencia múltiple. Una clase Java puede implementar múltiples interfaces Java. Todos los métodos en una interfaz son implícitamente públicos y abstractos.

1.2 Diferencias entre Interfaces y Clases Abstractas

- La diferencia principal es que los métodos de las interfaces son implicitamente abstractos y no pueden tener implementaciones. Una clase abstracta Java puede tener métodos de instancia que implementan un comportamiento por default.
- Todas las variables declaradas en una interface son de tipo final por default. Una clase abstracta puede contener variables que no sean final.
- Los miembros de una interface son publicos por default. En una clase abstracta los miembros pueden ser private, public, protected.
- Una interface deberia ser implementada utilizando la palabra reservada “implements”; una clase abstracta deberia ser heredada utilizando la palabra “extends”.
- Una interface puede heredar solo de otra interface, una clase abstracta puede heredar de otra clase e implementar multiples interfaces.
- Una clase puede implementar múltiples interfaces pero solo puede heredar de una clase padre.

1.3 Collection

Esta interfaz es “la raíz” de todas las interfaces y clases relacionadas con colecciones de elementos. Algunas colecciones pueden admitir duplicados de elementos dentro de ellas, mientras que otras no admiten duplicados. Otras colecciones pueden tener los elementos ordenados, mientras que en otras no existe orden definido entre sus elementos. Java no define ninguna implementación de esta

interface y son respectivamente sus subinterfaces las que implementarán sus métodos como son por ejemplo las interfaces Set o List (que son subinterfaces de Collection).

Una colección es de manera genérica un grupo de objetos llamados elementos. Esta interfaz por tanto será usada para pasar colecciones de elementos o manipularlos de la manera más general deseada.

En resumen la idea es la siguiente: cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos.

En Java, se emplea la interface genérica Collection para este propósito. Gracias a esta interface, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección, etc… ya que se trata de métodos definidos por la interface que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella.

Collection se ramifica en subinterfaces como Set, List y otras mas.

1.4 Tipos de colecciones

Java proporciona una serie de estructuras (interfaces, clases, etc) muy variadas para almacenar datos.

El siguiente esquema resume las interfaces y clases más usadas para colecciones de datos:

Un tipo de colecciones de objetos en Java son los Maps. A pesar de que estas estructuras denominadas “mapas” o “mapeos” son colecciones de datos, en el api de Java no derivan de la interface Collection. No obstante, muchas veces se estudian conjuntamente junto a las clases e interfaces derivadas de Collection.

El siguiente esquema resume cómo se organiza la interface Collection y sus derivaciones:

Estas estructuras ofrecen diversas funcionalidades: ordenación de elementos o no, mejor rendimiento para la ordenación para la inserción, tipos de operaciones disponibles, etc. Es importante conocer cada una de ellas para saber cuál es la estructura más adecuada en función de la situación en que nos encontremos. Por ejemplo si necesitamos realizar miles de búsquedas al día es importante elegir una estructura que tenga buenos rendimientos para búsquedas. Si necesitamos realizar miles de inserciones manteniendo un orden es importante usar una estructura que tenga buenos rendimientos para inserciones. Un buen uso de estas estructuras mejorará el rendimiento de nuestra aplicación. Para conocer qué tipo de colección usar, podemos emplear diagramas de decisión similares al siguiente:

Ejemplo

1.5 Tipos genéricos

Los generics son importantes ya que permiten al compilador informar de muchos errores de compilación que hasta el momento solo se descubrirían en tiempo de ejecución, al mismo tiempo permiten eliminar los cast simplificando, reduciendo la repetición y aumentando la legibilidad el código. Los errores por cast inválido son especialmente problemáticos de debuggear ya que el error se suele producir en un sitio alejado del de la causa.

Los generics permiten usar tipos para parametrizar las clases, interfaces y métodos al definirlas. Los beneficios son: - Comprobación de tipos más fuerte en tiempo de compilación. - Eliminación de casts aumentando la legibilidad del código. - Posibilidad de implementar algoritmos genéricos, con tipado seguro.

Los generics nos permiten manipular y operar con objetos sin especificar su tipo y posibilitan reutilizar código, además de ofrecer type safety, gracias al chequeo de tipos durante la compilación.

1.5.1 Type safety

Hace referencia a que el compilador valida los tipos mientras compila y lanza un error si se intenta asignar un tipo erroneo a una variable.

Por ejemplo: // Este bloque falla al intentar asignar un int a un String String one = 1; // también falla int foo = “bar”;

Esto también aplica a los argumentos de un método

int AddTwoNumbers(int a, int b) { return a + b; }

Al intentar llamar al método

int Sum = AddTwoNumbers(5, “5”);

El compilador lanzaria un error ya que estamos pasando un String (“5”) y está esperando un int.

1.5.2 Convención de nombres

Dado que se trata de tipos genéricos, su nomenclatura no afecta su comportamiento y podría designarse cualquier nombre para un genérico en Java. Sin embargo, existen convenciones para los nombres de estos tipos cuyo objetivo es mejorar la legibilidad e interpretación del código. Algunas de importancia son:

- E – Element.
- K – Key.
- V – Value.
- N – Number.
- T – Type.
- S, U, V, y así sucesivamente, para más tipos.

1.5.3 Ventajas (y desventaja)

Entre los pros de utilizar generics, podemos destacar:

- Permite la reutilización de código: es posible utilizar el mismo código para tipos diferentes.
- Evita el casteo de clases (por casteo entendemos realizar cambios entre tipos de datos distintos, donde chocan tipos de valores de datos).
- Código de mayor calidad, legible y limpio, si se utilizan las convenciones con criterio.
- Type safety: código más seguro, ya que reduce las oportunidades de que se introduzca un error.

- Rapidez de ejecución, dado que no hay chequeo de tipos en este tiempo. El casteo, en cambio, requiere que la JVM haga chequeo de tipos en tiempo de ejecución, en caso de que sea necesario ejecutar una ClassCastException.
- Chequeo de tipos en tiempo de compilación: el compilador verifica que se utilice la clase genérica donde se la invoca; en cambio, no lo hace cuando se invoca al tipo Object.
- Como contra, podemos señalar cierta dificultad en la interpretación del código si no se utilizan los nombres con claridad o no se cuenta con documentación.

Como contra, podemos señalar cierta dificultad en la interpretación del código si no se utilizan los nombres con claridad o no se cuenta con documentación.

1.5.4 Type erasure

Cuando se utilizan generics, se realizan casteos y comprobaciones de tipos en tiempo de compilación, los cuales tienen en cuenta la información almacenada en metadata para los tipos de generics. Por lo tanto, luego de que el compilador chequea las validaciones necesarias, elimina esa metadata en su totalidad. A este proceso de eliminación se lo denomina type erasure o borrado de tipos; no existe información de tipos en tiempo de ejecución y el uso de tipos para los generics se valida en tiempo de compilación.

1.5.5 Bounds

Incluso utilizando generics, a veces se presentan ciertas restricciones respecto de los tipos, teniendo que usar -en determinados casos- tipos concretos. Por ejemplo, en el caso de que una función aplique sólo a números, se utilizan los tipos límites o bounded types.

Con estos tipos, es posible limitar a los generics haciendo que se extiendan de una clase y en consecuencia, que su límite sea el de ese tipo.

Además, los bounds, en generics, pueden ser múltiples (límites múltiples). Estos representan la herencia múltiple de interfaces de Java para generics y, al igual que en la herencia común, es posible extender sólo de una clase y de varias interfaces.

La forma de utilizarlos es <T extends A & B1 & B2 & B3>. El orden en el que se colocan los límites es importante: si se extiende de una clase y de una -o varias- interfaces, la clase debe colocarse primero y luego las interfaces ya que, en caso contrario, habrá un error en la compilación. En este ejemplo, A sería una clase y Bn las interfaces.

Los bounds pueden utilizarse en el parámetro de una clase y en los parámetros que recibe y devuelve un método. En el caso del parámetro de una clase es importante resaltar que, por ejemplo, List`<A>` es distinta a List`<B>`, aunque A se extienda de B. De la misma manera, Box`<Number>` es distinto de Box`<Integer>`.

1.5.6 Wildcard types

Los tipos comodines o wildcard types en generics se utilizan para designar un tipo desconocido que puede ser de cualquier clase y será superclase para todas las demás. Se indica con el signo de interrogación "?". Por ejemplo: Collection`<?>`.

Pueden tener upper bounds o lower bounds para indicar que son subclases o superclases de otra clase y se indica de la siguiente forma: `<? extends someClass>`, para indicar de qué clase extiende y `<? super someClass>` para mostrar qué clase es superclase.

A diferencia de los casos anteriores, al utilizar estos tipos comodines, debemos tener en cuenta que, siguiendo el ejemplo de arriba, si bien Box`<Number>` es distinto de Box`<Integer>`, ambos extienden de Box`<?>`. Lo mismo ocurriría con las listas: List`<A>` y List`<B>` extienden de List`<?>`.

2 Patrones de Diseño en Java.

2.1 Introducción.

Christopher Alexander dice "Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para luego proponer el núcleo de la solución, de tal forma que esa solución puede ser usada millones de veces sin siquiera hacerlo dos veces de la misma forma".

En otra definición, "Los patrones de diseño son el esqueleto de las soluciones a problemas comunes en el desarrollo de software."

La siguiente es una lista de patrones de diseño con descripcipnes breves y con aplicaciones en ejemplos muy sencillos de entender.

2.2 Lista de Patrones de Diseño.

2.2.1 Patrones de Creación (Creational Patterns)

Relativos al proceso de creación de un objeto.

- Section 2.6
- Section 2.16.2
- Section 2.5
- Section 2.10
- Section 2.16.1

2.2.2 Patrones de Estructura (Structural Patterns)

Composición de clases u objetos.

- Section 2.11
- Section 2.7
- Section 2.12
- Section 2.13
- Section 2.15

2.2.3 Patrones de Comportamiento (Behavioral Patterns)

Forma en que clases las u objetos interaccionan y distribuyen funcionalidades.

- Section 2.14
- Section 2.4
- Section 2.3

2.3 Strategy Section ??

Propósito: Definir una familia de algoritmos, encapsular cada uno, y que sean intercambiables. Strategy permite al algoritmo variar independientemente de los clientes que lo utilizan.

Aplicación: Usamos el patrón Strategy cuando queremos… * Definir una familia de comportamientos. * Definir variantes de un mismo algoritmo. * Poder cambiar el comportamiento en tiempo de ejecución, es decir, dinámicamente. * Reducir largas listas de condicionales. * Evitar código duplicado. * Ocultar código complicado, o que no queremos revelar, del usuario.

Ejemplos: * Robot * Modos de transportación

2.4 Observer Section ??

Propósito: Defina una dependencia de uno a muchos entre los objetos de manera que cuando un objeto cambia de estado, todos los que dependen de él son notificados y se actualizan automáticamente.

Los *Observers* se registran con el *Subject* a medida que se crean. Siempre que el Subject cambie, difundirá a todos los Observers registrados que ha cambiado, y cada Observer consulta al Subject que supervisa para obtener el cambio de estado que se haya generado.

En Java tenemos acceso a la clase Observer mediante java.util.Observer

Aplicación: Usamos el patrón Observer cuando… * Un cambio en un objeto requiere cambiar los demás, pero no sabemos cuántos objetos hay que cambiar. * Configurar de manera dinámica un componente de la Vista, envés de estáticamente durante el tiempo de compilación. * Un objeto debe ser capaz de notificar a otros objetos sin que estos objetos estén fuertemente acoplados.

Ejemplos:

El ejemplo de Subasta presenta una variante distinta de muchas fuentes en Internet de este patrón, usaremos una clase *Evento* para controlar los sucesos a los que los *Observers* responderan.

- Subasta

2.5 Factory Section ??

Propósito: Definir una interface para crear un objeto, dejando a las subclases decidir de que tipo de clase se realizará la instancia. Reducir el uso del operador *new*.

Crear objetos en una clase usando un método factory es más flexible que crear un objeto directamente. Es posible conectar la generación de familias de clases que tienen comportamientos en común. Elimina la necesidad de estar haciendo binding (casting) hacia clases específicas dentro del código, ya que este solo se entiende con las clases abstractas.

Aplicación: Usamos el patrón Factory… * Cuando una clase no puede anticipar que clase de objetos debe crear, esto se sabe durante el tiempo de ejecución. * Cuando un método regresa una de muchas posibles clases que comparten carecterísticas comunes a través de una superclase. * Para encapsular la creación de objetos.

Ejemplos:

- Carros
- Naves

2.6 Abstract Factory Section ??

Propósito: Proveer una interfaz para la creación de familias o objetos dependientes relacionados, sin especificar sus clases concretas.

Es una jerarquía que encapsula muchas *familias* posibles y la creación de un conjunto de *productos*. El objeto "fábrica" tiene la responsabilidad de proporcionar servicios de creación para toda una familia de productos. Los "clientes" nunca crean directamente los objetos de la familia, piden la fábrica que los cree por ellos.

Aplicación: Usamos el patrón Abstract Factory… * Cuando tenemos una o múltiples familias de productos. * Cuando tenemos muchos objetos que pueden ser cambiados o agregados durante el tiempo de ejecución. * Cuando queremos obtener un objeto compuesto de otros objetos, los cuales desconocemos a que clase pertenecen. * Para encapsular la creación de muchos objetos.

Ejemplos:

- Ovnis
- El Reino

2.7 Composite Section ??

Propósito: Componer objetos en estructuras de árbol que representan jerarquías de un *todo* y sus *partes*. El Composite provee a los *clientes* un mismo trato para todos los objetos que forman la jerarquía.

Pensemos en nuestro sistema de archivos, este contiene *directorios* con *archivos* y a su vez estos *archivos* pueden ser otros *directorios* que contenga más *archivos*, y así sucesivamente. Lo anterior puede ser representado facilmente con el patrón Composite.

Aplicación: Usamos el patrón Composite… * Cuando queremos representar jerarquías de objetos compuestas por un todo y sus partes. * Se quiere que los *clientes* ignoren la diferencia entre la composición de objetos y su uso individual.

Ejemplos: * Menu * Sistema de Archivos * Cartas (Externo).

2.8 Singleton Section ??

Propósito: Asegurar que una clase tenga una única instancia y proporcionar un punto de acceso global a la misma. El *cliente* llama a la función de acceso cuando se requiere una referencia a la

instancia única.

Aplicación: Usamos el patrón Singleton… * La aplicación necesita una, y sólo una, instancia de una clase, además está instancia requiere ser accesible desde cualquier punto de la aplicación. * Tipicamente para: * Manejar conexiones con la base de datos. * La clase para hacer Login.

Ejemplos: * Gobierno

2.9 Builder Section ??

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

Nos permite crear un objeto que está compuesto por muchoso otros objetos. Sólo el "*Builder*" conoce a detalle las clases concretas de los objetos que serán creados, nadie más.

En este patrón intervienen un "*Director*" y un "*Builder*". El "*Director*" invoca los servicios del "*Builder*" el cual va creando las partes de un objeto complejo y al mismo tiempo guardo un estado intermedio de la construcción del objeto. Cuando el producto se ha construido por completo el *cliente* recupera el resultado.

A diferencia de otros patrones creacionales que construyen productos de una sola vez, el patrón "*Builder*" construye paso a paso los productos bajo el control del "*Director*".

Aplicación: Usamos el patrón Builder cuando queremos… * Construir un objeto compuesto de otros objetos. * Que la creación de las partes de un objeto sea independiente del objeto principal. * Ocultar la creación de las partes de un objeto del *cliente*, de esta manera no existe dependencia entre el *cliente* y las partes.

Ejemplos: * Heroes (basado en el siguiente proyecto) * Pizza * Robots * Comida rápida

2.10 Prototype Section ??

Propósito: Especificar varios tipos de objetos que pueden ser creados en un prototipo para crear nuevos objetos copiando ese prototipo. Reduce la necesidad de crear subclases.

Aplicación: Usamos el patrón Prototype… * Queremos crear nuevos objetos mediante la *clonación* o *copia* de otros. * Cuando tenemos muchas clases potenciales que queremos usar sólo si son requeridas durante el tiempo de ejecución.

Ejemplos: * Animales

2.11 Adapter Section ??

Como cualquier adaptador en el mundo real este patrón se utiliza para ser una interfaz, *un puente*, entre dos objetos. En el mundo real existen adaptadores para fuentes de alimentación, tarjetas de memoria de una cámara, entre otros. En el desarrollo de software, es lo mismo.

Propósito: Convertir la interfaz (**adaptee**) de una clase en otra interfaz (**target**) que el *cliente* espera. Permitir a dos interfaces incompatibles trabajar en conjunto. Este patrón nos permite ver a nuevos y distintos elementos como si fueran igual a la interfaz conocida por nuestra aplicación.

Aplicación: Usamos el patrón Adapter… * Cuando el *cliente* espera usar la interfaz de destino (**target**). * Deseamos usar una clase existente pero la interfaz que ofrece no concuerda con la que necesitamos.

Ejemplos: * Books

2.12 Decorator Section ??

Extender la funcionalidad de los objetos se puede hacer de forma estática en nuestro código (tiempo de compilación) mediante el uso de la herencia, sin embargo, podría ser necesario extender la funcionalidad de un objeto de manera dinámica.

Propósito: Adjuntar responsabilidades adicionales a un objeto de forma **dinámica**. Los *decoradores* proporcionan una alternativa flexible para ampliar la funcionalidad.

Aplicación: Usamos el patrón Decorator… * Cuando necesitamos añadir o eliminar dinámicamente las responsabilidades a un objeto, sin afectar a otros objetos. * Cuando queremos tener las ventajas de la *Herencia* pero ncesitemos añadir funcionalidad durante el tiempo de ejecución. Es más flexible que la *Herencia*, * Simplificar el código agregando funcionalidades usando muchas clases diferentes. * Evitar sobreescribir código viejo agregando, envés, código nuevo.

Ejemplos: * Pizzas

2.13 Facade Section ??

Propósito: Proporcionar una interfaz unificada para un conjunto de interfaces de un subsistema. *Facade* define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.

Detalles: * Este patrón protege los clientes de los componentes del subsistema, propiciando el menor uso de componentes para que el subsistema pueda ser utilizado. * Además, promueve un bajo acoplamiento entre subsistemas y clientes. * Este patrón no evita que los clientes usen las clases internas del subsistema, si es que es necesario. * Es importante mencionar que el objeto *Facade* debe ser extremadamente simple. **No** debe convertirse en un **objeto "dios".**

Aplicación: Usamos el patrón Facade… * Cuando queremos encapsular un subsistema complejo con una interface más simple. * Para crear una interface simplificada que ejecuta muchas acciones "detfas del escenario". * Existen muchas dependencias entre clientes y la implementación de clases de una abstracción. Se introduce el facade para desacoplar el subsistema de los clientes y otros subsistemas. * Necesitamos *desacoplar* subsistemas entre sí, haciendo que se comuniquen únicamente mediante *Facades*. * Para definir un punto de entrada a cada nivel del subsistema.

Ejemplos: * Banco * Computadora

2.14 Command Section ??

El patrón *Command* encapsula comandos(llamados a métodos) en objetos, permitiéndonos realizar peticiones sin conocer exactamente la petición que se realiza o el objeto al cuál se le hace la petición. Este patrón nos provee las opciones para hacer listas de comandos, hacer/deshacer acciones y otras manipulaciones.

Este patrón desacopla al *objeto que invoca* la operación del *objeto que sabe cómo* llevar a cabo la misma. Un objeto llamado *Invoker* transfiere el *comando* a otro objeto llamado *Receiver* el cual ejecuta el código correcto para el *comando* recibido.

Propósito: Encapsular una petición en forma de objeto, permitiendo de ese modo que parametrizar clientes con diferentes peticiones, "colas" o registros de solicitudes, y apoyar las operaciones de deshacer.

Aplicación: Usamos el patrón Command… * Cuando queremos realizar peticiones en diferentes tiempos. Se puede hacer a través de la especificación de una "cola". * Para implementar la función de deshacer (*undo*), ya que se puede almacenar el estado de la ejecución del comando para revertir sus efectos. * Cuando necesitemos mantener un registro (*log*) de los cambios y acciones.

Usos típicos: * Mantener un historial de peticiones. (*requests*) * Implementar la funcionalidad de *callbacks*. * Implementar la funcionalidad de *undo*.

Ejemplos: * Electrónicos * Hechizos

2.15 Proxy Section ??

Tengamos en cuenta el siguiente escenario: Es necesario instanciar objetos sólo cuando sean efectivamente solicitados (*request*) por el cliente.

Un **Proxy** o sustituto: 1. Crea una instancia del objeto real la primera vez que el cliente realiza una solicitud del proxy. 2. Recuerda la identidad de este objeto real. 3. Finalmente, envía la solicitud del servicio al objeto real.

Propósito: * Proveer un sustituto o *"placeholder"* de otro objeto para controlar el acceso a este. * Usar un nivel extra de indirección para permitir el acceso distribuido, controlado e inteligente. * Agregar un *"wrapper"* para proteger el componente real de la complejidad innecesaria. Este *wrapper* permite agregar funcionalidad al objeto de interés sin cambiar el código del objeto.

Aplicación: Usamos el patrón Proxy… * Cuando haya necesidad de una referencia más versátil y sofisticada a un objeto, no un simple puntero. * Para adicionar seguridad al acceso de un objeto. El Proxy determinará si el cliente puede acceder al objeto de interés. * Para proporcionar una API simplificada para que el código del cliente no tenga que lidiar con la complejidad del código del objeto de interés. * Para proporcionar una interfaz de los *web services* o recursos *REST*.

Escenarios de uso: * *Remote Proxy*: Representa un objeto local que pertenece a un espacio de direcciones diferente. * *Virtual Proxy*: En lugar de un objeto complejo o pesado, utiliza una representación de esqueleto. Consideremos una imagen la cual es enorme en tamaño, podemos representarla mediante un objeto proxy virtual y cuando sea solicitado cargamos el objeto real. * *Protection Proxy*: Controla el acceso al objeto original. Este tipo es útil cuando se necesita manejar diferentes permisos de acceso.

Ejemplos: * Images * ATM

2.16 Ejemplos:

2.16.1 Singleton

En ingeniería de software, singleton o instancia única es un patrón de diseño que permite restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.

Su intención consiste en garantizar que una clase solo tenga una instancia y proporcionar un punto de acceso global a ella.

Ejemplo

2.16.2 Builder

El patrón de diseño Builder permite crear objetos que habitualmente son complejos utilizando otro objeto más simple que los construye paso por paso.

Este patrón Builder se utiliza en situaciones en las que debe construirse un objeto repetidas veces o cuando este objeto tiene gran cantidad de atributos y objetos asociados, y en donde usar constructores para crear el objeto no es una solución cómoda.

Se trata de un patrón de diseño bastante útil también en la ejecución de test (unit test por ejemplo) en donde debemos crear el objeto con atributos válidos o por defecto.

Normalmente resuelve el problema sobre decidir qué constructor utilizar. A menudo las clases tienen muchos constructores y es muy difícil mantenerlos. Es común ver constructores múltiples con distintas combinaciones de parámetros.

Ejemplo

2.17 Maven

Maven es una herramienta open-source, que se creó en 2001 con el objetivo de simplificar los procesos de build (compilar y generar ejecutables a partir del código fuente). Antes de que Maven proporcionara una interfaz común para hacer builds del software, cada proyecto solía tener a alguna persona dedicada exclusivamente a configurar el proceso de build. Además, los desarrolladores tenían que perder tiempo en aprender las peculiaridades de cada nuevo proyecto en el que participaban. Si queríamos compilar y generar ejecutables de un proyecto, teníamos que analizar qué partes de código se debían compilar, qué librerías utilizaba el código, dónde incluirlas, qué dependencias de compilación había en el proyecto… En el mejor de los casos, se empleaban unos pocos minutos para saber cómo hacer una build del proyecto. En el peor de los casos, el proceso de build era tan complejo que un desarrollador podía tardar horas en saber cómo compilar y generar los ejecutables a partir del código. Ahora, la build de cualquier proyecto Maven, independientemente de sus módulos, dependencias, librerías…consiste simplemente en ejecutar el comando mvn install. Por otra parte, antes de Maven, cada vez que salía una nueva versión de un analizador estático de código, de un framework de pruebas unitarias (como JUnit) o cualquier librería, había que parar todo el desarrollo para reajustar el proceso de build a las nuevas necesidades. Y… ¿cómo se ejecutaban las pruebas? ¿Cómo se generaban informes? Sin Maven, en cada proyecto esto se hacía de distinta manera. Pero Maven, es mucho más que una herramienta que hace builds del código.

Como vemos, Maven simplifica mucho el proceso de build del código, permitiéndonos compilar cualquier tipo de proyecto de la misma manera, librándonos de todas las dificultades que hay por detrás. Pero lo cierto es que Maven es mucho más que una herramienta que hace builds del código. Podríamos decir, que Maven es una herramienta capaz de gestionar un proyecto software completo, desde la etapa en la que se comprueba que el código es correcto, hasta que se despliega la aplicación, pasando por la ejecución de pruebas y generación de informes y documentación. Para ello, en Maven se definen tres ciclos de build del software con una serie de etapas diferenciadas. Por ejemplo el ciclo por defecto tiene las etapas de: - Validación (validate): Validar que el proyecto es correcto. - Compilación (compile). - Test (test): Probar el código fuente usando un framework de pruebas unitarias. - Empaquetar (package): Empaquetar el código compilado y transformarlo en algún formato tipo .jar o .war. - Pruebas de integración (integration-test): Procesar y desplegar el código en algún entorno donde se puedan ejecutar las pruebas de integración. - Verificar que el código empaquetado es válido y cumple los criterios de calidad (verify). - Instalar el código empaquetado en el repositorio local de Maven, para usarlo como dependencia de otros proyectos (install). - Desplegar el código a un entorno (deploy). Para poder llevar a cabo alguna de estas fases en nuestro código, tan solo tendremos que ejecutar mvn y el nombre de la fase (la palabra que puse entre paréntesis). Además van en cadena, es decir, si empaquetamos el código (package), Maven ejecutará desde la fase de validación (validate) a empaquetación (package). Así de simple. Por otra parte, con Maven la gestión de dependencias entre módulos y distintas versiones de librerías se hace muy sencilla. En este caso, solo tenemos que indicar los módulos que componen el proyecto, o qué librerías utiliza el software que estamos desarrollando en un fichero de configuración de Maven del proyecto llamado POM. Además, en el caso de las librerías, no tienes ni tan siquiera que descargarlas a mano. Maven posee un repositorio remoto (Maven central) donde se encuentran la mayoría de librerías que se utilizan en los desarrollos de software, y que la propia herramienta se descarga cuando sea necesario.

Digamos que Maven aporta una semántica común al proceso de build y desarrollo del software. Incluso, establece una estructura común de directorios para todos los proyectos. Por ejemplo el código estará en raíz del proyecto/src/main/java, los recursos en raíz del proyecto/src/main/resources. Los tests están en raíz del proyecto /src/test etc.

Ejemplo

[ ]: