Desarrollador Fullstack Senior
Tópicos para entrevista

Jorge Rivera
jorge.rivera@ssde.com.mx

July 2024

Índice general

Parte I

Generales

1 Patrones de diseño

Un patrón de diseño es una solución general repetible a un problema que ocurre comunmente. No es un diseño terminado que pueda convertirse directamente a código. Es una descripción o plantilla de cómo resolver un problema que puede ser utilizado en muchas situaciones diferentes.

Usos de patrones de diseño

Los patrones de diseño pueden acelerar el proceso de desarrollo a través de paradigmas de desarrollo probados y comprobados. El diseño de software efectivo requiere considerar problemas que pueden no ser visibles sino hasta tarde en la implementación. Reusar patrones de diseño ayuda a prevenir problemas sutiles que pueden generar problemas mayores y mejora la legibilidad del código para desarrolladores y arquitectos familiarizados con los patrones.

Los patrones de diseño comunes pueden ser mejorados iterativamente haciendolos más robustosque diseñnos a medida.

Los patrones pueden dividirse en 3 tipos

- Patrones creacionistas,
- Patrones estructurales y
- Patrones de comportamiento

1.1. Patrones creacionistas

Éstos patrones se tratan de instanciación de clases . Pueden dividirse a su vez en patrones de creación de clases y patrones de creación de objetos. Mientras que los primeros usan la herencia efectivamente en el proceso de instanciación, los segundos usan la delegación efectivamente para completar su trabajo.

Algunos ejemplos de éste tipo de patrón son:

- **Fabrica abstracta (Abstract Factory):**
  Crea una instancia de varias familias de clase. El cliente de software usa un interfaz genérica de la fabrica para crear objetos concretos que son parte de la familia. El cliente no sabe que objetos concretos recibe de cada una de éstas fábricas internas porque usa sólo las interfaces genéricas de sus productos

- **Constructor (Builder):**
  Separa la representación de un objeto de su implementación. Éste patrón a diferencia de tros patrones creacionistas no requiere que se tengan interfaces comunes. Ésto hace posible crear diferentes productos usango el mismo proceso de construcción. Éste patrón puede reconocerse en una clase que tiene un solo metodo para crearla y varios metodos para configurar el objeto resultante. Lo objetos de éste patrón comunmente soportan el encadenado. Ex:

```
someBuilder.setValueA(1).setValueB(2).create();
```

- **Método de Fábrica (Factory Method):**
  Éste patrón de diseño provee una forma de crear objetos pero permite a las subclases alterar el tipo de objeto que será creado. Si tenemos un método de creación en la clase base subclases que la extienden, pudieramos estar hablando de éste patrón. Ex:

```
abstract class Department {
 public abstract function createEmployee($id);
```

```
 public function fire($id) {
        $employee = $this->createEmployee($id);
        $employee->paySalary();
        $employee->dismiss();
    }
}

class ITDepartment extends Department {
 public function createEmployee($id) {
  return new Programmer();
 }
}

class AccountingDepartment extends Department {
 public function createEmployee($id) {
  return new Accountant();
 }
}
```

- **Alberca de objetos (Objects pool):**
  Provee una alberca de objetos pre-inicializados que pueden ser reutilizados en lugar de crearlos y destruirlos bajo demanda. Ésto mejora el desempeño y reduce el sobre-trabajo de crear y destruir objetos

- **Prototipo (Prototype):**
  Las clases prototipo deben tener una interfaz común que haga posiblecopiar objetos aún si sus clases concretas son desconocidas. Los objetos prototipo puedenn producir copias completas debido a que los objetos de la misma clase pueden acceder a las propiedades privadas una de la otra.

- **Singleton:**
  En éste patrón de diseño sólo puede existir una instancia de cada clase.

1.2. Patrones estructurales

Éstos patrones se tratan de composición de clases y objetos. Éstos patrones de creación de clases usan la herencia para componer interfaces. Se definen formas de acomodar objetos para obtener nueva funcionalidad.

Algunos ejemplos de éste tipo de patrón son:

- **Adaptador (Adapter):** Empata interfaces con distintas clases. Permite que objetos con distintas interfaces colaborar.
- **Puente (Bridge):** Éste es un patrón que nos permite dividir un clase grande o un conjunto de clases relacionadas estrechamente en 2 jerarquías separadas *abstracción* e *implementación*, las cuales pueden desarrollarse independientes una de otra.

  En éste patrón los términos abstracción e implementación no son los mismos que utilizamos en nuestros lenguajes de programación, éstos se definen a continuación:

  + **Abstracción:** Es la capa de control de más alto nivel. Ésta capa no se supone que haga ningún tipo de trabajo. Ésta debe solo delegar el trabajo a la capa de implementación (plataforma).
  + **Implementación:** Ésta se refiere a la capa donde se procesan nuestras peticiones de la capa anterior, por ejemplo una API o un servicio.
- **Compuestos (Composed):** Éste patrón nos permite componer objetos en estructura de árbol y luego trabajar con éstas estructuras como si fueran objetos individuales. Más detalles aquí
- **Decorador (Decorator):** Permite adjuntar nuevos comportamientos a objetos al ponerlos dentro de objetos envoltorio que contienen dichos comportamientos. Más detalles aquí
- **Facade:** Provee una interfaz simplificada a una librería, framework o algún otro conjunto de clases complejas. Más detalles aquí

- **Peso mosca (flyweight):** Permite acomodar más objetos en la cantidad disponible de RAM compartiendo partes de su estado entre múltiples objetos en lugar de mantener todos los datos en cada objeto. Más detalles aquí
- **Datos de clase privada (Private data class):** Restringe el acceso al estado interno de un objeto mejorando la seguridad y reduciendo los riesgos de corrupción de datos por medio de métodos de acceso controlados
- **Proxy:** Provee un sustituto o ancla para otro objeto. Un proxy controla el acceso al objeto original permitiendonos realizar acciones ya sea antes o después de que el requerimiento llega al objeto original. Más detalles aquí

1.3. Patrones de comportamiento

Éstos patrones se tratan de la comunicación de clases de objetos. Éstos patrones son aquellos que están más específicamente preocupados por la comunicación entre objetos.

Algunos ejemplos de éstos patrones son:

- **Cadena de responsabilidad (Cahin of Responsibility):** Permite pasar requerimientos a través de una cadena de manejadores. Al recibir el requerimiento cada manejador decide ya sea procesarlo o pasarlo al siguiente manejador en la cadena. Más detalles aquí

2 AWS

3 Docker/Kubernates

Parte II

Desarrollo Back End

4 Java Core

Java fue desarrollado originalmente por James Gosling para Sun Microsystems en Mayo de 1995 como un componente base de la plaaforma de Java de Sun. Las librerias, compiladores y máquinas virtuales originales fueron lanzadas bajo licencias propietarias de Sun. A partir de 2007 en cumplimiento con las nirmas de Java Community Process, Sun cambió su licencia de la mayría de sus tecnologías Java a la licencia GPL-2.0-only. Oracle tiene su propia implementación de máquina virtual, sin embargo la versión oficial es la de OpenJDK la cual es gratuita, de código abierto y la más utilizada por los desarrolladores y es la versión defacto utilizada por casi todas las distribuciones de Linux.

A continuación se detallan varias particularidades de éste lenguaje.

4.1. Tipos de dato

Java no es un LPOO completo porque tenemos tipos de datos primitivos, ie; int, long, boolean, double así como sus clases contenedor Integer, Long, Double, Boolean.

En un LPOO todos los elementos son objetos, no hay tipos primitivos.

4.2. Modificadores de acceso

Todos los objetos en Java pueden restringirse usando los modificadores public, private protected y default, éste último no es necesario utilizarlo, al no utilizar ningún modificador se toma éste por defecto, a continuación se definen los alcances de los diferentes modificadores.

- **default:** El alcance de éste modificador es a nivel de paquete, esto es, cualquier objeto dentro del mismo paquete puede acceder a la propiedad o método.
- **private:** Éste modificadores el más restrictivo, permite acceso solo a los elementos de la clase a la cual pertenecen.

- **protected:** Permiten acceso a elementos del mismo paquete o sus subclases inclusive si éstas no están dentro del mismo paquete.
- **public:** Éste es el más permisivo de todos, permite acceso desde cualquier clase o metodo dentro de la aplicación

Los modificadores son importantes porque nos permiten:

- **Encapsulamiento:** Nos permiten encapsular codigo en clases y exponer solo las partes del código que requiere acceso. Ésto reduce la dependencia entre clases.
- **Previene mal uso:** Previene mal uso de metodos y propiedades en formas que no se tienen planificadas.
- **Seguridad:** Restringir el acceso a datos y metodos sensibles mejora la seguridad porque esconde detalles de implementación a potenciales atacantes.
- **Refactorización:** El código que utiliza modificadores apropiados es más fácil de simplificar porque reducir la visibilidad no rompe otros código.
- **Legibilidad:** Hacen el código más facil de leer porque explícitamente se asume quienes pueden acceder a los componentes.
- **Reuso:** Las clases que hacen buen uso de los modificadores serán más fáciles de reutilizar en otros proyectos sin modificaciones extensas.
- **Control de Interfaz:** Permiten definir interfaces publicas estables para clases mientras se mantiene la implementación privada

Para mayor información referente a modificadores de acceso y su implementación visite la siguiente liga.

4.3. Genéricos

Fueron introducidos en Java 1.5, se buscaba reducir los errores y adicionar una capa extra de abstracción a los tipos.

Imaginemos que tenemos el código siguiente:

```
List list = new LinkedList();
list.add(new Integer(1));
Integer i = list.iterator().next();
```

Sorprendentemente el compilador nos mandará un error en la última línea porque no sabe el tipo de dato que se regresa. El compilador necesita un cast explícito:

```
Integer i = (Integer) list.iterator.next();
```

Nada nos garantiza que el tipo de dato devuelto por la lista es un *Integer*, la lista que definimos puede contener objetos de cualquier tipo. Nosotros solo sabemos que estamos recibiendo una lista al inspeccionar el contexto. Cuando vemos los tipos, solo podemos garantizar que es un objeto y por lo tanto se necesita de un cast explícito para asegurar que el tipo es el adecuado.

Hacer cast es engorroso - sabemos que el tipo de dato en la lista en un *Integer*. El cast también nos infla el código y puede generar errores en tiempo de ejecución relacionados a tipos de datos si un programador comete un error con la conversión de tipos.

Lo más sencillo es que los programadores expresen sus intenciones de utilizar un tipo específico de dato y que el compilador se asegure de la correctez de dichos tipos. Ésta es la idea detrás de los genéricos.

Modifiquemos el código para incluir los genericos:

```
List<Integer> list = new LinkedList<>();
```

Al agregar el operador diamante <> conteniendo el tipo de dato, especializamos la lista para solo utilizar el tipo *Integer*. En otras palabras, especificamos el tipo de dato que la lista contiene. El compilador entonces puede forzar el tipo al tiempo de compilar.

En programas pequeños puede verse trivial pero para programas más grandes ésto puede agregar robustez y hacer el código fácil de leer.

4.4. Map, HashMap, LinkedHashMap, TreeMap, ConcurrentHashMap

- **Map:** Es una interfaz con una correspondencia clave-valor
- **HashMap:** Es un Map que utiliza una hash table para su implementación. Permite nulos en claves o valores
- **HashTable:** Es una versión sincronizada de HashMap. No permite nulos en claves o valores.
- **TreeMap:** Usa un árbol para implementar un Map. Ordena los elementos de acuerdo a un iterador, si no se especifica uno se oprdenan naturalmente en orden ascendente
- **ConcurrentHashMap:** Permite a varios hilos que lo accedan al mismo tiempo y de forma segura
- **LinkedHashMap:** Conserva el orden de iteración de los objetos que fueron insertados (otros no proporcionan un orden de iteración fijo)

Ésta misma diferencuación puede hacerse con las colecciones List y Set.

5 Cambios entre versiones

A través de los años éste lenguaje ha recibido una considerable cantidad de mejoras, desde su introducción en el año 1995 hasta la fecha.

Java ganó mucha popularidad desde su lanzamiento y ha sido popular desde entonces. En 2022 era el 3er lenguaje más popular según Github. Aunque es vastamente utilizado, ha habido un declive en su uso en años recientes.

Hasta Marzo de 2024, Java 22 es la última version. Java 8, 11, 17 y 21 son las versiones LTS que todavía tienen soporte.

5.1. Cambios en Java 8

Ésta versión fue la más esperada actualización de Java debido a que en toda la historia de éste lenguaje no habían sido liberadas tantos cambios significativos. Ésta liberación se dió el 18 de Marzo de 2014. Ésta nueva versión vino acompañada de soporte para programación funcional, nuevas APIs, un nuevo motor de JavaScript y otros cambios que serán detallados a continuación

https://www.digitalocean.com/community/tutorials/java-8-features-with-examples

5.1.1. Expresiones lambda

Éstas permiten una representación concisa de funciones anónimas, habilitando paradigmas de programación funcional en Java.

Una expresión lambda consiste en una lista de parámetros, un token de flecha $\rightarrow$ y un cuerpo. Son particularmente útiles para implementar interfaces de un solo método (interfaces funcionales) y para pasar comportamiento como parámetro a un método.

Éstas expresiones ayudan a reducir codigo común y para hacer el código más legible y mantenible.

5.1.2. Interfaces funcionales

Se introduce la anotación $@FunctionalInterface$ para definir las interfaces con un solo metodo abstracto aunque no es obligatorio, se pueden tener varios métodos default y estáticos dentro de la interfaz. Éstas interfaces facilitan el uso de expresiones lambda y referencias de metodo, habilitando los patrones de programación funcional.

5.1.3. forEach e interfaces iterables

Cada vez que necesitamos recorrer una colleción es necesario crear un iterador cuyo propósito es simplemente recorrer la colección y luego tenemos

la lógica en un ciclo por cada uno de los elementos en dicha colección.

5.1.4. Métodos default

5.1.5. Métodos estáticos

5.1.6. Referencias a métodos

5.1.7. Introducción y mejora de APIs

Con la llegada de ésta nueva versión llegaron varias APIs que eran muy utilizadas y eran de terceros y otras que se adoptaron de otros lenguajes

5.1.7.1. Introducción de API Stream

Ésta API se introdujo para procesar operaciones filter/map/reduce con colecciones de objetos con el estilo de código funcional usando expresiones lambda. Para entender lo que es la API Stream debes tener conocimiento de ambos, expresiones lambda e interfaces funcionales.

Ésta API también permite ejecuciones paralelas y secuenciales. Ésta es una de las mejores características porque permite trabajar con con colecciones y Big Data por las operaciones de filtrado que se ejecutan con regularidad. Para utlizar éstas características tenemos las funciones stream() y parallelStream().

Las colecciones difieren en varias formas de los streams, por una parte, las colecciones son estructuras que se almacenan en memoria, la cual almacena sus valores. Cada elemento en la colección debe ser calculado antes de ser almacenado en la misma. Las operaciones de búsqueda, ordenamiento, inserción manipulación y borrado pueden realizarse en las colecciones. Cuenta con muchas interfaces como Set, List, Queue, Deque y varias clases como ArrayList, Vector LinkedList, PriorityQueue y HashSet.

Por otro lado, Stream es una API, es una secuencia de objetos que usan varios metodos que pueden ser ligados para obtener el resultado deseado. Más información aquí

5.1.7.2. Introducción de API Date/Time

Siempre ha sido difícil trabajar con Fechas, Horas y Zonas horarias en Java. No había una forma estándar para realizar éstas operaciones. Una de las mejores adiciones en Java 8 fue el paquete java.time que aceleró el proceso de trabajar con horas y fechas en java.

Simplemente con ver los paquetes de ésta API, podemos inferir que será muy fácil de utilizar. Tiene algunos subpaquetes java.time.format que provee clases para imprimir y dar formato a fechas y horas y el subpaquete java.time.zone que provee ayuda con las reglas de las zonas horarias.

Ésta nueva API favorece el uso de enums sobre las constantes enteras para meses y dias de la semana. Una de las clases útiles es DateTimeFormatter para convertir objetos DateTime en cadenas. Más información aquí

5.1.7.3. Mejora de API Collection

Ya vimos el método forEach y la API Stream para las colecciones, también hay otras adiciones a la API Collections, las cuales detallaremos a continuación:

- **forEachRemaining(Consumer action):** Es un iterador que realiza la acción dada para cada elemento remanente en la colección hasta que todos los elementos hayan sido procesados o la acción lance una excepción.
- **removeIf(Predicate filter):** Es un método que elimina todos los elementos que cumplen con la condición dada.
- **spliterator():** Es un método que devuelve una instancia Spliterator que puede ser utilizada para recorrer elementos secualcial o paralelamente.
- **replaceAll(), compute(), merge():** Métodos para la interfaz Map
- **HashMap:** Mejoras en el desempeño con colisiones de claves (Key Collisions)

5.1.7.4. Mejora de API Concurrency

Las mejoras más importantes son:

- **ConcurrentHasMap:** Nuevos métodos compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() y search().
- **CompletableFuture:** Acciones que pueden ser explicitamente completadas (asignando sus valores y estado).
- **newWorkStealingPool():** Método ejecutpor que crea un hilo robatrabajos que utiliza todos los procesadores disponibles como su nivel objetivo de paralelismo.

5.1.8. Clase optional

Todo programador está familiarizado con *NullPointerException*. Ésto puede descarrilar el código y es difícil de evitar sin utlizar muchas revisiones de nulos. Es así como en Java 8 se introdujo una clase Optional al paquete java.util. Ésta nos ayuda a escribir código sencillo sin utilizar tantas revisiones de nulos. Al utilizar Optional podemos especificar valores alternativos de retorno o asignar código alternativo para ejecutar. Ésto hace el código más legible porque situaciones que estaban ocultas están ahora visibles al desarrollador.

Más información aquí

5.2. Cambios en Java 11

5.3. Cambios en Java 17

5.4. Cambios en Java 21

5.5. Collections

5.5.1. Lists

5.5.2. Sets

5.5.3. Maps

5.6. Streams

6 Hibernate

7 Spring Boot

7.1. Anotaciones

7.1.1. @Autowired

7.1.2. @Repository

7.1.3. @Service

7.1.3.1. @Primary

7.1.3.2. @Qualifier

7.1.4. @Controller

7.1.5. @RestController

7.1.5.1. @RequestMapping

7.1.5.2. @RequestBody

7.1.5.3. @PathVariable

7.1.5.4. @GetMapping

7.1.5.5. @PostMapping

7.1.5.6. @PutMapping

7.1.5.7. @PatchMapping

7.1.5.8. @DeleteMapping

7.1.6. @ControllerAdvice

7.1.6.1. @ExceptionHandler

7.2. JPA

7.2.1. Anotaciones

7.2.1.1. @Entity

7.2.1.2. @Table

7.2.1.3. @Id

7.2.1.4. @GeneratedValue

7.2.1.5. @OneToOne

7.2.1.6. @OneToMany

7.2.1.7. @ManyToMany

7.2.1.8. @JoinColumn

7.2.1.9. @JoinTable

7.2.2. Interfaces Repositorio

7.3. Pruebas

7.3.1. Unitarias

7.3.2. Integración

7.4. Seguridad

7.4.1. OAuth2

7.4.2. OKTA

7.4.3. JWT

Parte III

Desarrollo Front End

8 Angular

8.1. CLI

8.2. Estructura de una aplicación

8.3. Modulos

8.4. Servicios

8.5. Componentes

8.5.1. Compartiendo datos entre componentes

- Padre a hijo: @Input
- Hijo a padre: @Output and EventEmitter
- Hijo a padre: @ViewChild
- Componentes no relacionados: Usando servicios

9 React