Desarrollo web en entorno servidor

UT1.2 – Programación web

2 – Inversión de control (IoC) – Inyección de dependencias (DI)

IES Clara del Rey – Madrid

Dependencias en las clases

Cuando estamos programando una clase esta puede depender de otras clases o interfaces.

Estas clases o interfaces pueden ser del mismo proyecto o de una dependencia externa configurada con classpath, Maven o Gradle.

Por ejemplo, si queremos generar datos aleatorios con DataFaker (datafaker.net), tendríamos que añadir la dependencia Maven o Gradle, y luego usar la clase en nuestro código.

Si creamos nuestro propio generador de datos aleatorios, será parte de nuestro programa, pero también será una dependencia, pero interna.

No hay diferencias relevantes entre las dependencias internas (código propio) o externas (código ajeno), salvo su origen.

Ejemplo de dependencia

Supongamos un sistema en el que tenemos:

- Clase "UserService", para operaciones relacionadas con usuarios.
- En concreto, hay una operación "RegisterUser" que:
  - Crea un usuario en el sistema, en estado "pendiente de validar"
  - Envía un correo al usuario, para que valide la dirección.
- Para el envío del correo, la clase "UserService" usa la clase "MailService", que tiene métodos para el envío de correo.
- Hay una dependencia entre UserService y MailService. UserService depende de MailService para poder hacer su trabajo. Necesita un objeto de la clase MailService.

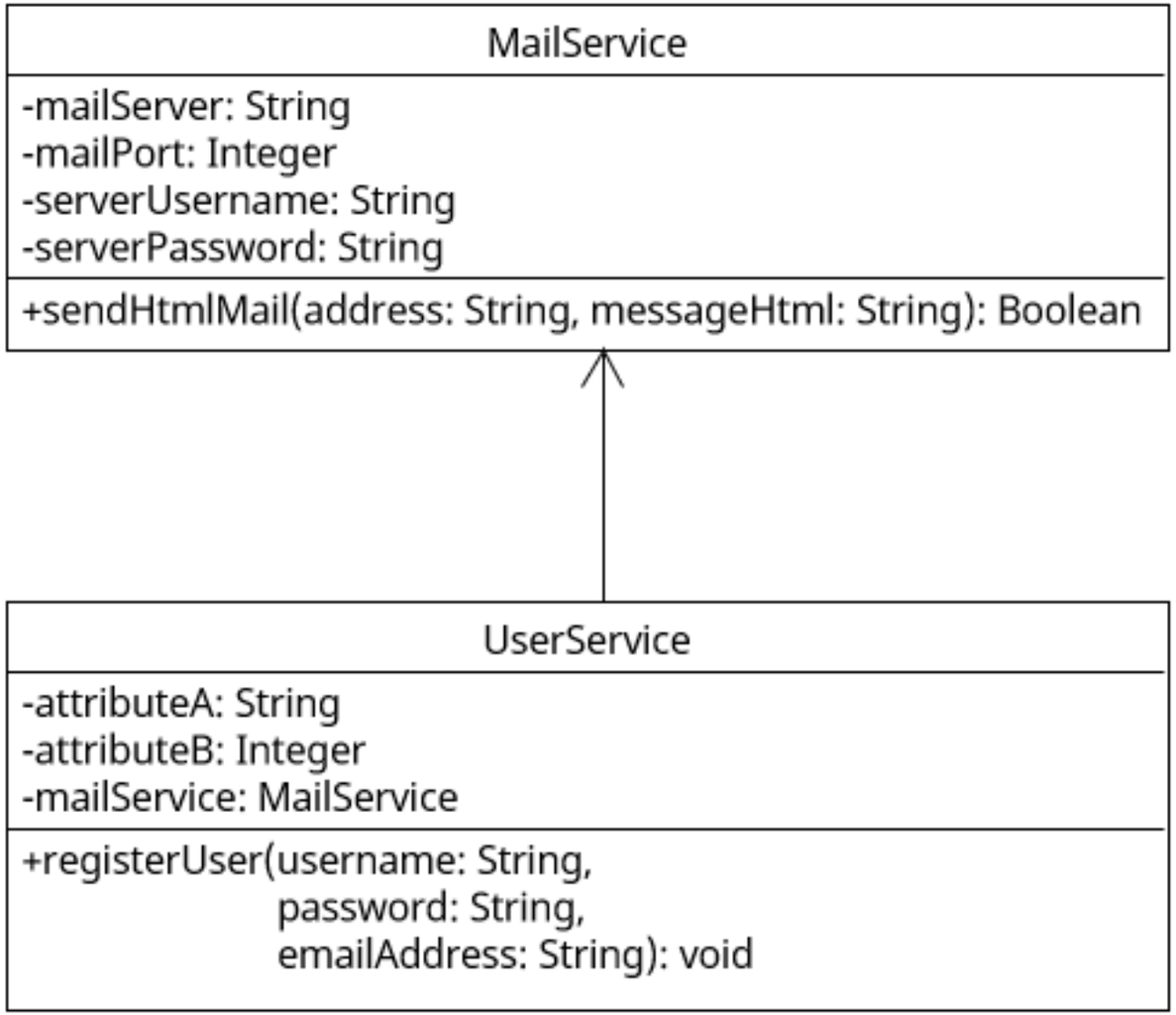
Ejemplo de dependencia

Esto, en UML, se representaría de forma similar a esta.

UserService depende de MailService.

La dependencia se produce porque UserService tiene un atributo privado de tipo MailService.

UserService utiliza este objeto en el método sendHtmlMail, para realizar el envío.



Control "clásico" de la dependencia

Con lo que sabemos hasta ahora, en la mayoría de los casos, la propia clase "UserService" es la responsable de crear el objeto para satisfacer la dependencia.

Por ejemplo, en la declaración del atributo.

```
private MailService mailService = new MailService();
```

O en el constructor

```
private MailService mailService;
public UserService() {
        this.mailService = new MailService();
}
```

Control "clásico" de la dependencia

Este modelo de control "clásico", en el que la clase dependiente crea los objetos de los que depende, desde el punto de vista de arquitectura de software, tiene algunos inconvenientes:

- Alto acoplamiento: la clase dependiente está muy acoplada con las clases de las que depende. No se puede sustituir la dependencia sin modificar la clase dependiente. No podemos cambiar MailService por SecureMailService sin cambiar el código de UserService
- Dificultad para probar código independientemente. Para probar la clase dependiente tenemos que, a la vez, probar la dependencia. No podemos probar el método registerUser de UserService sin, a la vez, probar el método sendHtmlMail de mailService.

Control "clásico" de la dependencia

- Poca flexibilidad: la creación de objetos internos hace que sea complicado reutilizar la clase en otros contextos, ya que está "atada", "condicionada" a las dependencias que ha creado.
- Responsabilidad multiplicada - Violación del SRP (Responsabilidad Única): Al crear dependencias, la clase asume una responsabilidad adicional, crear las dependencias. Lo ideal sería que la creación de dependencias fuera responsabilidad de otro.
- Difícil configuración y gestión de dependencias en proyectos grandes: a medida que los proyectos crecen hay más complejidad, y mantener dependencias, modificarlas, implica cambios en cascada en el código fuente, que pueden extenderse y provocar errores.

Inversión de control (IoC)

Principio de diseño en que se invierte la forma "convencional" o "clásica" de gestionar las dependencias.

La clase dependiente ya no crea los objetos de los que depende.

¿Quién los crea? Pues depende de cada framework de desarrollo, pero lo habitual es que se encargue un componente denominado "contenedor de dependencias" (Dependency container) o "contenedor de inyección de dependencias" (DI container) o "contenedor de inversión de control" (IoC container).

Este contenedor es el encargado de crear las dependencias en el momento que son necesarias, y gestionar su ciclo de vida: cuando se crean, cuando se asignan a clases que dependen de ellas, cuando se destruyen, etc.

Inversión de control (IoC) – Puntos clave

- Creación e inyección: el contedor IoC realiza dos tareas:
  - Creación de las dependencias, de los objetos.
  - Pasar los objetos creados a las clases que dependen de ellos. Este paso de instancias se denomina "Inyección de dependencias" (DI)
- Cambio en el quien controla las dependencias:
  - Sin IoC, cada clase debe crear los objetos de los que depende.
  - Con IoC, el contenedor es responsable de "inyectar" las dependencias en las clases dependientes cuando son necesarias.
- Desacoplamiento: Separar creación e inyección, reduce el acoplamiento, y facilita el reemplazo de dependencias con otras implementaciones. Por ejemplo, para realizar pruebas.

Inyección de dependencias (DI)

Este patrón de diseño permite que los objetos reciban sus dependencias desde el exterior en lugar de crearlas.

En general hay tres tipos de inyección de dependencias, en función de la forma en que el contenedor IoC inyecta las dependencias en las clases:

- Inyección de atributos: El contendor IoC asigna valores directamente a los atributos inyectables de una clase.
- Inyección por setter: El contendor IoC usa un setter del atributo inyectable para asignarle valor.
- Inyección por constructor: El contenedor IoC pasa las dependencias al constructor de la clase dependiente, que las guarda en atributos. El más recomendado, porque es el que hace el código más testeable.

Otras formas de inversión de control

Además de la inyección de dependencias, hay otras formas de implementar la IoC