Spring Framework

Dependency Injection Övningar

Mål

I denna övning kommer vi att:

- Undersöka hur en implementation kan skapa stark koppling mellan klasser.
- Lära oss att minska koppling genom att använda:
    - Programmera mot interface
    - Dependency Injection

Koppling i ett exempelprojekt

Java version och IDE

- Använd Java 17 för detta projekt.
- Välj IntelliJ som utvecklingsmiljö.

Ladda ner projektet

Ladda ner *1-beginof-coupling.zip*, och importera projektet *Coupling Example* i Intellij.

Projektets struktur

- Det finns en klass som heter *Invoice* - en enkel Java-klass (POJO).
- En annan klass, *InvoicingDAO*, hanterar databasanslutning och sparar fakturor.
- Projektet använder *JDBC* för att kommunicera med databasen.
- Databasen *HSQLDB* används eftersom den skapas i minnet och sparas till en fil när programmet avslutas.
- En Service-klass fungerar som en mellanhand för att undvika direkt koppling mellan klient och databas.
- Efter att ha kört klientkoden, kontrollera projektet och notera de nya filerna
    - *invoicedb.dat.script*

    - *invoicedb.dat.properties.*
- Öppna skriptfilen och se sammanhanget. Det finns en insert-sats i filen:
    - INSERT INTO INVOICES VALUES('11075','Bill McLarane')

Problem med nuvarande implementation

- Ju större applikationen blir, desto svårare blir det att göra ändringar.
- Svårt att enhetstesta serviceklasser.

Lösningen är att minska kopplingen genom att:

- Programmera mot gränssnitt.
- Använda Dependency Injection.
- Centralisera konfigurationen (tas upp i nästa övning).

Steg 1: Programmera mot Interface

- Byt namn på klassen *InvoicingDAO* till *InvoicingDAOJdbcImplementation*.
- Och deklarera att den här klassen implementerar *InvoicingDAO* interfacet
- Skapa ett interface vid namn *InvoicingDAO*.
- Deklarera tre metoder i interfacet:
    - save(Invoice newInvoice)
    - delete(Invoice oldInvoice)
    - update(Invoice invoiceToCancel)
- Skapa en ny implementation *HibernateDAO* som också implementerar *InvoicingDAO*.
- Se till att varje metod har en enkel utskrift för att verifiera att rätt implementering används.

Steg 2: Ändringar i InvoiceService

- Ersätt direkt skapande av *InvoicingDAOJdbcImplementation* med interfacet *InvoicingDAO*.
- Ersätt alla referenser till *InvoicingDAOJdbcImplementation* med *InvoicingDAO*.
- Testa koden och verifiera att den fortfarande fungerar.
- Notera att vi fortfarande har koppling eftersom vi använder ‘*new’* för att skapa objekt.

Steg 3: Dependency Injection

- I *InvoiceService*, skapa ett privat fält *dao* av typen *InvoicingDAO*.
- Skapa en setter-metod som tar emot ett InvoicingDAO-objekt och tilldelar det till *dao.*
- Ta bort all kod som använder *new* för att skapa dao i *raiseInvoice* metoden.
- I klientklassen:
    - Skapa en instans av InvoicingDAO (dao) och instansiera den till *InvoicingDAOJdbcImplementation*.
    - Anropa setter metoden för att skicka in *dao* i *InvoiceService*.
    - Ändra datan för den nya invoice till:
        - Invoice newInvoice = new Invoice("11076", "Susan McLarane");
    - Testa koden och verifiera att den fungerar.
        - Öppna skriptfilen och se sammanhanget. Det finns en till insert-sats i filen:

INSERT INTO INVOICES VALUES('11076','Susan McLarane')

Nästa steg

Om vi vill byta från JDBC till Hibernate skulle det innebära en mindre ändring i klientkoden, där vi bara byter implementation. Nu har vi fått en viss flexibilitet, men vi har fortfarande koppling i vår klientkod. Därför bör vi ta nästa steg, vilket är att centralisera konfigurationen, och det är här Spring kommer in i bilden.

I nästa kapitel kommer vi att titta närmare på containrar och specifikt på Spring.