# Dependency Injection

## Ein Beispiel welches das Problem zeigt

Stellen Sie sich vor, Sie hätten eine Liste von Musikalben in einer Datei. Jedes Album ist definiert durch seinen Titel und den Namen der Band, welche das Album eingespielt hat.

Ihre Aufgabe ist es nun, die Liste zu filtern, so dass nur die Alben einer vorgegebenen Band ausgegeben werden. – That's easy, Isn't it?

### Klasse «Album»

```
public class Album {
    private String band;
    private String title;

    public Album(String band, String title) {
        this.band = band; this.title = title;
    }

    public String getBand()  { return band; }
    public String getTitle() { return title; }
}
```

### Klasse «AlbumFinderFromFile»

```
public class AlbumFinderFromFile {
    private String fileName;

    public AlbumFinderFromFile(String fileName) { this.fileName = fileName; }

    public List<Album> getAllAlbums() {
        return null; // TODO : Should read the records from a file
    }
}
```

### Klasse «AlbumListerFromFile»

```
public class AlbumListerFromFile {
    private AlbumFinderFromFile reader = new AlbumFinderFromFile("./albums.txt");

    public List<Album> findAlbumsFromBand(String band) {
        List<Album> result = new ArrayList<>();
        for(Album album : reader.getAllAlbums()) {
            if (album.getBand().equals(band)) { result.add(album); }
        }
        return result;
    }
}
```

Aber was haben wir genau gemacht?

## Lösung mittels Komposition

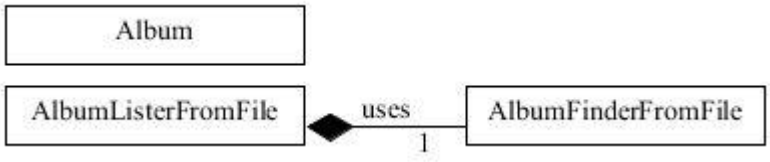


Abbildung 1 Assoziationsdiagramm

Mit dem Erstellen eines «AlbumListerFromFile»-Objektes entsteht automatisch auch ein «AlbumFinderFromFile»-Objekt.

Problem gelöst, oder? Mmm ja, eigentlich nein.

Stellen Sie sich vor, dass Sie eine weitere Liste von Musikalben haben, diesmal aber in einer DB gespeichert. Kein Problem sagen Sie, ich mache einfach eine neue Klasse «AlbumFinderFromDb». OK, eine perfekte Idee, aber jetzt müssen Sie auch eine neue Klasse «AlbumListerFomDb» schreiben, und das alles nur weil «AlbumFinderFromFile» zu «AlbumFinderFromDb» geändert hat. Irgendwie keine gute Idee.

## Lösung mittels Aggregation

Wie immer in der Informatik, fragen wir uns, was bleibt konstant und was ändert sich.

Die Klasse «AlbumLister» bleibt konstant. Die «AlbumFinder»-Klassen ändern sich. Wir erfinden also wie üblich ein Interface, dass die Änderungen abstrahiert, und implementieren dann einfach passende Klassen für den File- bzw. Datenbankzugriff.

Der Klasse «AlbumLister» wird dann ein passendes «AlbumFinder»-Objekt im Konstruktor übergeben.

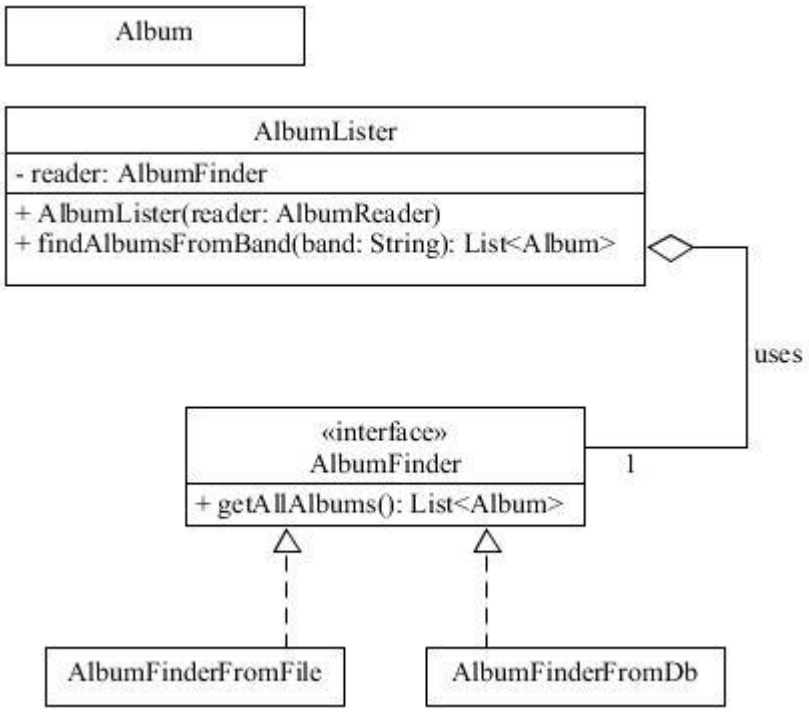


Abbildung 2 Klassendiagramm (Aggregation)

Eigentlich eine elegante Lösung. Die «AlbumLister» Klasse existiert nur einmal und kann mit einem beliebigen «AlbumFinder» –Objekt parametrisiert werden. Wenn wir später eine Albumliste aus dem Netz lesen wollen, implementieren wir einfach eine neue Klasse «AlbumFinderFromWeb» welche «AlbumFinder» implementiert.

Einen kleinen Hacken hat die Sache aber noch. Irgendwo muss festgelegt werden, welcher «AlbumFinder» aktuell verwendet werden soll.

```
public class Main {
    public static void main(String[] args) {
        AlbumLister finder =
       new AlbumLister(new AlbumFinderFromFile("./albums.txt"));
        System.out.println(finder.findAlbumsFromBand("Motörhead"));
    }
}
```

## Lösung mittels Dependency Injection

Dependency Injection bedeutet, dass wir auf den letzten Schritt (die explizite Parametrierung des «AlbumLister»-Objektes) verzichten können. Je nach verwendetem Framework fällt auch der Konstruktor für die Parametrierung weg. Wir müssen nur angeben, dass wir unser Field mit einem passenden Objekt bestückt haben wollen, den Rest erledigt dann der Toolkit.

In Spring werden dazu die Membervariablen mittels der Annotation «@AutoWired» ausgezeichnet.

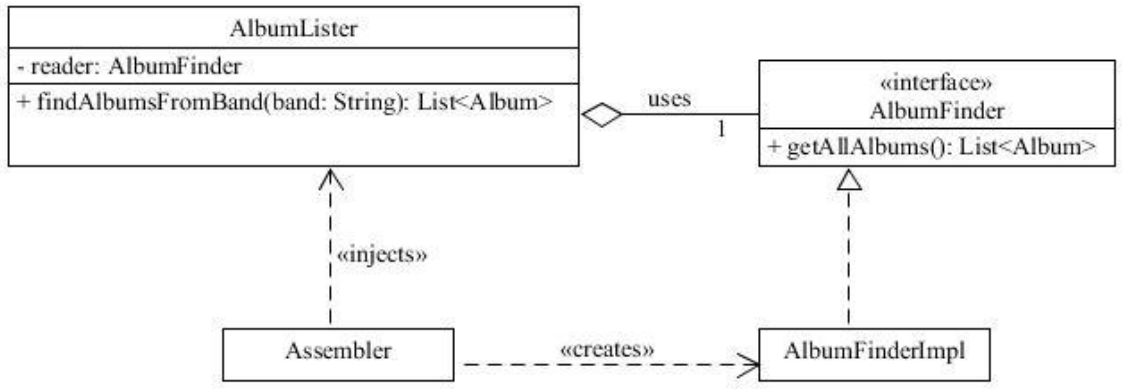


Abbildung 3 Klassendiagramm (Dependencys Injection)

Spring übernimmt hier die Rolle des Assemblers.