Dependency Injection

Von David Kofler, einige Ergänzungen von Matthias Dieter Wallnöfer

Der Großteil moderner Software nutzt eine Vielzahl verschiedenster Anwendungskomponenten (Klassen, Packages, Bibliotheken), welche gemeinhin als *Abhängigkeiten/Dependencies* bezeichnet werden. Dabei ist es nicht unüblich, dass Letztere wiederum auf Dienstleistungen weiterer Komponenten zurückgreifen. Dies ermöglicht eine möglichst ökonomische Entwicklung sehr umfangreicher Programme, bei der das Rad nicht zweimal neu erfunden werden muss. Andererseits sind damit auch größere Risiken[1] verbunden, so dass die bedarfsgerechte Auswahl mit gewissenhafter Sorgfalt vorgenommen werden sollte. Neben der Verfügbarkeit einer API-Dokumentation und Qualität des Codes sollte beispielsweise auch der Prozess zur Bewältigung auftretender Sicherheitslücken Beachtung finden (Gibt es Sicherheitsverantwortliche?).

Das *Single Responsibility Principle* (**SRP**)[2] besagt, dass sich jeder Software-Baustein nur für eine Aufgabe zuständig sieht: Beispielsweise sollte die individuell anpassbare Newsletter-Komponente nicht eine eigene Implementierung für das SMTP-Protokoll vorsehen. Solcher Code befindet sich auf einer anderen Abstraktionsebene, wobei er genau so gut in einem E-Mail Client Verwendung finden könnte. Dem SRP folgend wird die Komplexität einzelner Komponenten reduziert und im besten Fall ihre Wiederverwendbarkeit verbessert. Ferner wird es möglich, für eine Vielzahl kleinerer Module umfassende Unit-Tests zu entwickeln und im Anschluss daran deren Integration gewissenhaft zu testen.

Abhängigkeiten bzw. Komponenten-Hierarchien lassen sich mit fester Verdrahtung im Quellcode umsetzen. Dieser einfache, aber unflexible Ansatz mag oft passend erscheinen, jedoch kann er später auftretende Änderungswünsche enorm einschränken. Eleganter geht es mit dem *Injektionsmuster*, welches folgender Definition Genüge tut:

> *Das **Dependency Injection-Pattern** (DI) beschreibt die Idee, dass erforderliche Abhängigkeiten einer Komponente nicht von der Komponente selbst erzeugt bzw. entfernt werden. Dies geschieht außerhalb ihres Kontrollbereichs, wobei sie weiterhin vollen Zugriff auf alle ihre Abhängigkeiten genießt.*
> *Einfacher ausgedrückt: Die Komponente besitzt nicht ihre Kinder, sondern sie leiht sie sich aus.*

Arten von Dependency Injection

Setter Injection

```
class Abhängiges {
  private Abhängigkeit abhängigkeit;

  public void setAbhängigkeit(final Abhängigkeit abhängigkeit) {
    this.abhängigkeit = abhängigkeit;
  }
}

class Injizierer {
  void methode() {
    final Abhängiges abhängiges = ... ;
```

1 Lücken in SW-Paketen wie jenen im *npm* (node.js Package Manager) bzw. im Java-Logger *log4j* (Ende 2021)
2 Auch in Unix bzw. GNU/Linux gültig

```
    final Abhängigkeit abhängigkeit = ... ;
    abhängiges.setAbhängigkeit(abhängigkeit);
  }
}
```

Der einfachste Ansatz besteht in der Übergabe der Abhängigkeiten in Form von Setter-Methoden. Neben dem Vorteil, Dependencies gewissermaßen als *optional* zu deklarieren, zieht er große Nachteile mit sich:

- Es kann sein, dass das Objekt erst nach der Ausführung einer gewissen Anzahl von Methoden *vollständig initialisiert* ist. Somit ist es nicht unrealistisch, dass für eine schlecht programmierte Komponente die Setter-Methoden in einer gewissen Reihenfolge aufgerufen werden müssen.
- Die Komponente kann sich niemals sicher sein, ob sie bereits über die gewünschte Abhängigkeit verfügt. Es kann jederzeit eine *„NullPointerException“* auftreten, nicht nur nach unterlassener Initialisierung, sondern auch später durch einen Setter-Aufruf mit Parameter *null*.

Diese Mängel lassen sich mit entsprechenden Absicherungen im Quellcode lindern, allerdings steht das Konstrukt auf wackeligen Beinen: Es braucht nur eine weitere Abhängigkeit dazukommen, welche es erneut beeinträchtigt.

Field Injection

Ferner besteht die Möglichkeit, Felder des Objekts als öffentlich zu deklarieren und das Übergeben der Dependencies von außen zu gestatten. Der Nachteil ist jener, dass durch diese Technik das Prinzip der *Kapselung gewissermaßen mit Füßen* getreten wird.
Überhaupt treffen alle Nachteile der Setter-Injektion auch hier zu und zwar *ohne* die Möglichkeit, das Setzen der Felder in irgendeiner Weise zu überwachen.

Constructor Injection

```
class Abhängiges {
  private Abhängigkeit abhängigkeit;

  public Abhängiges(final Abhängigkeit abhängigkeit) {
    this.abhängigkeit = abhängigkeit;
  }
}

class Injizierer {
  void methode() {
    final Abhängigkeit abhängigkeit = ... ;
    final Abhängiges abhängiges = new Abhängiges(abhängigkeit) ;
  }
}
```

Bei dieser Technik werden die Dependencies über den Konstruktor in ein neues Objekt der Komponente geladen, um im Anschluss privaten Feldern zugewiesen zu werden. Pro injiziertem Objekt wird *ein* Argument des Konstruktors benötigt, was dessen Parameterliste recht schnell in die Länge wachsen lässt.

Ein weiterer Nachteil besteht in der *Kombination voneinander abhängiger Komponenten*. Dazu folgendes Szenario aus der Welt der Datenbank-Frameworks (ORM) gegriffen: Der Konfigurationslader benötigt die Datenbankkomponente, während die Datenbankkomponente Konfigurationswerte benötigt.

In gewisser Hinsicht ist die Schwierigkeit, zirkulare Abhängigkeiten zu entwerfen, eher von Vorteil: Üblicherweise deuten diese auf Designprobleme hin und sollten a priori vermieden werden.

Interface-Injection

```
interface IInjizierbar {
  void injiziere(final Abhängigkeit abhängigkeit);
}

class Abhängiges implements IInjizierbar {
  private final Abhängigkeit abhängigkeit;

  public void injiziere(final Abhängigkeit abhängigkeit) {
    this.abhängigkeit = abhängigkeit;
  }
}

class Injizierer {
  void methode() {
    final Abhängigkeit abhängigkeit = ... ;
    final IInjizierbar injizierbares = ... ;
    injizierbares.injiziere(abhängigkeit);
  }
}
```

Im Unterschied zur Konstruktor-Injektion bedient man sich einer vordefinierten Schnittstelle, über die Klassen zur Laufzeit die notwendigen Abhängigkeiten zur Verfügung gestellt bekommen. Hierbei ist es fundamental sicherzustellen, dass bei allen Komponenten, die die Schnittstelle implementieren, dann auch wirklich die *Injektionsmethode zur Ausführung gelangt*.

Einsatz

Ein mithilfe von Dependency Injection strukturiertes Programm verfügt über eine Hauptmethode, die das Laden der Konfiguration, das Initialisieren der Komponenten und die Zuweisungen (d.h. Injektionen) vornimmt. Heutzutage wird dieser Code nicht mehr selber programmiert und der Programmierer bekommt ihn auch nicht zu Gesicht.

Viele Frameworks (auch Angular) bieten die Möglichkeit an, mithilfe von Annotations- bzw. Konfigurationsdateien die Struktur der fertigen Anwendung vorzugeben (bei durchgehender Verwendung von Constructor Injection ist diese ein Baum) und sie dann automatisch zu initialisieren. Dem Programmierer bleibt damit nur mehr die Aufgabe überlassen, die gewünschten Komponenten zu importieren, um sie abschließend dem Framework bekannt zu machen.

Es bietet sich an, die Klasse samt Hauptmethode und spezialisierten Komponenten in ein eigenes Modul zu geben, damit bei Änderungen am Anwendungscode nicht alle Anwendungsmodule ausgetauscht werden müssen. Dabei wird man von intelligenten *Package-Management-Buildsystemen* wie Maven (Java), npm (node.js), NuGet (C#) oder auch PyPI (Python) unterstützt.

- Definition Wikipedia: https://de.wikipedia.org/wiki/Dependency_Injection
- Liste bekannter DI-Frameworks: https://de.wikipedia.org/wiki/Liste_von_Dependency_Injection_Frameworks