Labor: Dependency Look-Up und -Injection

1.1.1 Inversion of Control bei der Erzeugung von Objekten

Wenn ein Objekt von einem anderen Objekt abhängig ist, besitzt es in der Regel eine Referenz auf das andere Objekt, um es für seine Zwecke bei Bedarf nutzen zu können. Allgemein gesprochen sind diese beiden Objekte miteinander verknüpft. Um ein Objekt mit einem anderen Objekt zu verknüpfen und damit eine Abhängigkeitsbeziehung herzustellen, gibt es mehrere Möglichkeiten. Die offensichtlichste ist, dass ein Objekt dasjenige Objekt, das es benötigt, **selbst** erzeugt, wodurch dieses Objekt erstens Informationen über das andere Objekt halten muss und außerdem diese Abhängigkeit statisch im Quellcode festgelegt werden muss. Durch Verwendung einer **Fabrikmethode** (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**) kann bei der Erzeugung eines benötigten Objektes etwas Flexibilität gewonnen werden. Aber auch in einer Fabrikmethode wird ein Objekt erzeugt und die Klasse des zu erzeugenden Objekts statisch im Quellcode der entsprechenden Unterklasse festgelegt.

Inversion of Control bezüglich der Erzeugung von Objekten bedeutet nun, dass die Kontrolle darüber, welches Objekt wann erzeugt wird, von der nutzenden Klasse abgegeben wird. Hierfür werden die beiden folgenden Techniken verwendet:

- **Dependency Look-Up**
  Bei einem Dependency Look-Up sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt z. B. in einem Register, um die Verknüpfung mit diesem herzustellen. Damit hängt es nicht von jedem anderen Objekt ab, das es benötigt, sondern nur vom Register.
- **Dependency Injection**[1]
  Hierbei wird die Erzeugung von Objekten und die Zuordnung von Abhängigkeiten an eine dritte Partei delegiert. Damit sind die Objekte untereinander nicht abhängig.

1.1.1.1 Dependency Look-Up

Bei einem Dependency Look-Up sucht ein Objekt, das ein anderes Objekt braucht, nach diesem anderen Objekt, um die Verknüpfung mit diesem herzustellen. Die einzige Abhängigkeit zwischen den beiden Objekten ist, dass das suchende Objekt den logischen Namen des gesuchten Objekts kennen muss, allerdings aber nicht mehr die Details über das gesuchte Objekt wie beispielsweise dessen Klasse.

[1] Der Begriff Dependency Injection wurde zum ersten Mal von M. Fowler in [fowioc] benutzt, um diese Technik von der Inversion of Control abzugrenzen.

Dependency Look-Up bedeutet, dass ein Objekt seine **Verknüpfung** mit einem anderen Objekt **zur Laufzeit** herstellt, indem es noch diesem anderen Objekt über einen logischen Namen zur Laufzeit sucht.

Das gesuchte Objekt kann selber wieder verknüpft sein. Ob diese weiteren Verknüpfungen bereits hergestellt sind oder nicht, hängt von der Anwendung ab. Prinzipiell kann die Suche rekursiv fortgesetzt werden, bis alle Verknüpfungen aufgelöst sind. Dadurch können die Verknüpfungen sehr flexibel hergestellt werden, was beispielsweise bei Objekten von Vorteil ist, die von Frameworks erzeugt werden, aber noch Verknüpfungen zu den Objekten der Anwendung benötigen.

Zur Unterstützung eines Dependency Look-Up gibt es mehrere Möglichkeiten:

- Die einfachste Möglichkeit hierzu ist, die benötigten Objekte in einer **zentralen Klasse** zu halten, auf die alle anfragenden Objekte Zugriff haben. Wird ein Objekt oder eine Beziehung über den logischen Namen in dieser zentralen Klasse gesucht, erhält das suchende Objekt alles, was es braucht. Es braucht also de facto nur den speziellen, logischen Namen eines anderen Objekts zu kennen.
- Eine weitere Möglichkeit besteht im Einsatz einer **Registratur** (engl. **registry**). Die erzeugten Objekte werden registriert, also in eine Registratur eingetragen. Wird ein Objekt für eine Verknüpfung benötigt, wird bei der Registratur über den logischen Objektnamen nach einem entsprechenden Objekt gesucht. Die Registratur liefert eine Referenz auf das gesuchte Objekt zurück, womit dann eine Verknüpfung hergestellt werden kann.
- Eine häufig eingesetzte Lösung ist es, alle in einer Anwendung benötigten Objekte in einem **Container**-Objekt als Behälter zu halten und nur die Referenz auf dieses Container-Objekt allen anfragenden Objekten zur Verfügung zu stellen. Im Programmcode können dann über dieses Container-Objekt die benötigten Objekte abgefragt werden[2]. Je nach Implementierung des Containers kann entweder ein Programmierer dafür zuständig sein, das Container-Objekt mit Objekten zu füllen und ggf. Verknüpfungen zwischen ihnen herzustellen, oder das Container-Objekt ist – beispielsweise mit Hilfe von Properties-Dateien (siehe Kapitel 1.1.2) – selbst in der Lage, die benötigten Objekte und ihre Verknüpfungen zu erzeugen. Die Objekte erhalten dabei ihre Verknüpfungen zu den zugehörigen Objekten von dem Container-Objekt als Dienstleistung. Damit dieser Dienst genutzt werden kann, muss er den Objekten bekannt sein und damit sind die Objekte von diesem Dienst abhängig. Allerdings sind sie untereinander unabhängig, was ja das Ziel dieses Ansatzes ist.

[2] Dies kann mehr oder weniger als eine Verallgemeinerung des Musters **Objektpool** (siehe Kapitel 4.21) gesehen werden, da die in einem Container-Objekt gespeicherten Objekte gänzlich unterschiedliche Typen haben können.

Aufgabe 1: Dependency Look-Up mit Hilfe einer Registry

In Aufgabe 1 soll ein Plotter betrachtet werden. Die Aufgabe des Plotters ist es, eine Reihe von Daten, die von einer Datenquelle erzeugt werden, aufzubereiten und grafisch darzustellen. Die Klasse `Plotter` aggregiert das Interface `IDatenquelle`, welches auf verschiedene Arten realisiert werden kann. In diesem Beispiel wird das Interface `IDatenquelle` von der Klasse `DatenquelleImpl` realisiert. Alle vorhandenen Realisierungen einer Datenquelle werden in einer Registratur verwaltet. Die Klasse `Plotter` verwendet die Registratur, um eine konkrete Datenquelle auszuwählen, damit die darin enthaltene Datenreihe dann dargestellt werden kann. Bitte ersetzen Sie die Lücken . . . durch entsprechenden Code.

Hier das Interface `IDatenquelle`:

```
// Datei: IDatenquelle.java
import java.awt.Point;
import java.util.ArrayList;

public interface IDatenquelle
{
   public ArrayList<Point> getDatenreihe();
}
```

In der Klasse `DatenquelleImpl` wird eine konkrete Datenreihe beispielsweise aus Zufallszahlen generiert:

```
// Datei: DatenquelleImpl.java
// ...

public class DatenquelleImpl implements IDatenquelle
{
   public ArrayList<Point> getDatenreihe()
   {
      // Implementierung einer Datenreihe, nicht abedruckt

   }
}
```

Die Klasse `Registratur` dient in dieser Aufgabe der Verwaltung aller existierenden Datenquellen. Bei der Klasse `Registratur` soll das **Singleton-Muster** eingesetzt werden, da von dieser Klasse nur ein einziges Objekt benötigt wird. Die Klasse `Registratur` kann nicht nur Datenquellen verwalten, sondern es können bei ihr beliebige Objekte registriert werden. Nachfolgend der Lückencode der Klasse `Registratur`:

```
// Datei: Registratur.java
// Die Klasse Registratur kann beliebig viele Objekte verwalten.
// Jedes Objekt wird durch einen String-Schluessel registriert und
// kann durch diesen angefordert werden.
import java.util.HashMap;

public class Registratur
{
   private Registratur(){};

   private HashMap<String, Object> objekt =
                new HashMap<String, Object>();
   private static Registratur registratur=null;

   public static Registratur getRegistratur ()
   {
      ...
   }

   public Object getObjekt(String objektBezeichnung)
   {
      ...
   }

   public void registriereObjekt(String bezeichnung, Object objekt)
   {
      ...
   }
}
```

In der folgenden Klasse `Plotter` soll die grafische Ausgabe einer Datenreihe generiert werden.

```
// Datei: Plotter.java
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Point;
import java.util.ArrayList;

import javax.swing.JFrame;
import javax.swing.JPanel;

public class Plotter extends JPanel
{
   // Die Klasse Plotter besitzt eine Referenz auf eine Datenquelle
   private IDatenquelle datenquelle;
   . . .
   public void plot()
```

```
   {
      // Datenquelle von der Registratur anfordern
      ...

      // Wenn keine Datenquelle in der Registratur vorhanden ist
      // beende den Plotvorgang
      ...

      // Durch den Aufruf der Methode setVisible() wird veranlasst,
      // dass die Methode paintComponent() aufgerufen wird und
      // damit das Fenster gezeichnet wird.
      frame.setVisible(true);
   }

   protected void paintComponent(Graphics g)
   {
      ...
   }
}
```

Die `main()`-Methode der Klasse `TestPlotter` soll eine Datenquelle erzeugen und diese dann unter dem Namen `"Datenquelle"` in der Registratur registrieren. Der Plotter, der anschließend gestartet werden soll, kann über diesen Namen auf die Datenquelle zugreifen und dieses Objekt benutzen. Hier der Lückencode der Klasse `TestPlotter`:

```
// Datei: TestPlotter.java
public class TestPlotter
{
   static public void main(String[] args)
   {
      // Bekomme Referenz auf die Registratur
      ...

      // Registriere eine Datenquelle als Objekt
      ...

      // plotten ...
      ...
   }
}
```

Das folgende Bild zeigt eine vom Plotter gezeichnete Datenreihe:

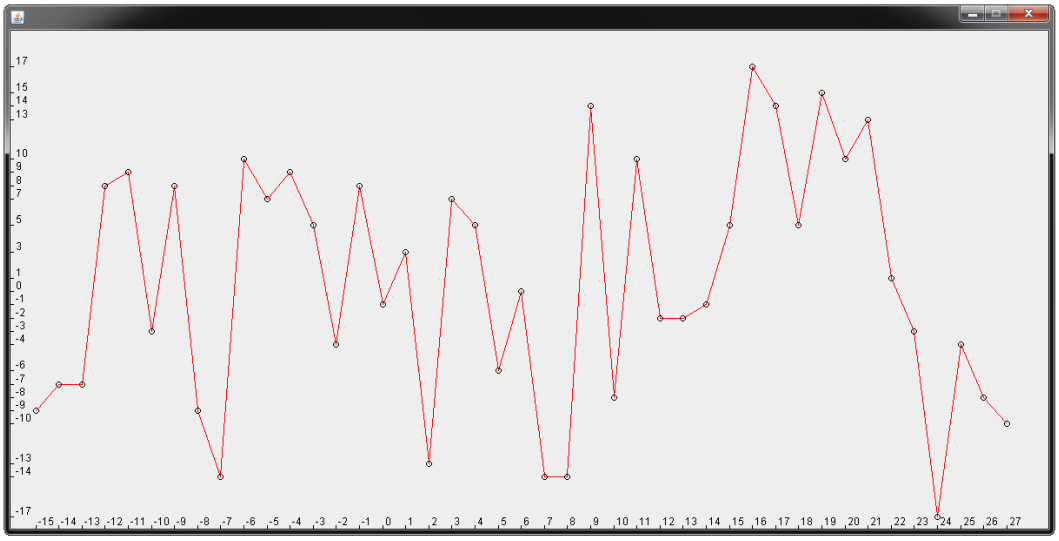

*Bild **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**-1 Geplottete Datenreihe*

1.1.2 Dependency Injection

Der Ansatzpunkt von Dependency Injection ist der gleiche wie beim Dependency Look-Up: Direkte Abhängigkeitsbeziehungen zwischen miteinander verknüpften Objekten bzw. Klassen sollen vermieden werden.

Bei Dependency Injection werden Abhängigkeiten **von außen** übergeben (injiziert). Ein Objekt sucht nicht aktiv wie beim Dependency Look-Up in einer Registry, sondern bleibt passiv, was die Verknüpfung betrifft.

Die Objekte erhalten also Referenzen auf Objekte, die sie benötigen, von außen – von einer eigenen Instanz, dem Injektor. Ein Injektor erzeugt alle Objekte und Verbindungen. Der Programmcode eines Objekts bzw. seiner Klasse ist daher von den anderen Klassen entkoppelt. Es bestehen zur Kompilierzeit keine Abhängigkeiten von den anderen Klassen. Erst zur Laufzeit werden die Verbindungen zwischen den einzelnen Objekten durch den Injektor hergestellt. Auf die Realisierungsformen von Dependency Injection sowie auf die Implementierungsmöglichkeiten des Injektors wird im Verlauf des Kapitels noch detailliert eingegangen.

Dependency Injection bedeutet, dass eine **Verknüpfung** zwischen Objekten **zur Laufzeit** von einer eigenen Instanz (**Injektor**) hergestellt wird und nicht zur Kompilierzeit.

Vorteile von Dependency Injection

Durch Dependency Injection wird eine Klasse unabhängig von anderen Klassen und braucht auch kein Wissen darüber, wie Objekte solcher Klassen zu erzeugen sind. Wenn Klassen keine Abhängigkeiten zu anderen Klassen haben, sind sie als isolierte Elemente auch am einfachsten zu testen. Man kann eine Klasse also isolieren, wenn ihre Objekte keine Objekte von anderen Klassen zu erzeugen brauchen, sondern ihnen Referenzen auf die benötigten Objekte zur Laufzeit mittels Dependency Injection übergeben werden.

Realisierungsformen der Dependency Injection

Üblicherweise erzeugt in objektorientierten Systemen ein Objekt **die von ihm benötigten** Objekte selbst. Dadurch besitzt es auch eine Referenz auf das jeweilige erzeugte Objekt. Bei Dependency Injection überträgt man die Aufgabe für das Erzeugen und für das Verknüpfen von Objekten an eine eigene Instanz wie etwa ein Framework.

Für Dependency Injection gibt es nach [fowioc] folgende drei Möglichkeiten:

- Es können Abhängigkeiten zu anderen Objekten direkt bei der Erzeugung über einen Konstruktor mit Parametern erzeugt werden (**Constructor Injection**).
- Es ist aber auch denkbar, dass diese erst später vom Injektor über set-Methoden festgelegt werden (**Setter Injection**).
- Die dritte Technik ist, Interfaces zu definieren und für das Injizieren zu verwenden (**Interface Injection**).

Während die ersten beiden Möglichkeiten – also Constructor Injection und Setter Injection – offensichtlich sind, bedarf die Vorgehensweise mittels Interface Injection einer ausführlichen Betrachtung.

Bei Interface Injection wird zusätzlich zu der Abstraktion für die zu injizierenden Objekte (`IAbstraktion`) noch ein weiteres Interface (`IInjectAbstraktion`) für die Injektion dieser Objekte vorgegeben, siehe Bild **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**-2. Die Klasse eines Objekts, in das eine Verbindung injiziert werden soll, muss das Interface `IInjectAbstraktion` implementieren. Das Interface `IInjectAbstraktion` definiert eine Methode `inject()`, die von einem Injektor zur Laufzeit aufgerufen werden kann, um die Verbindung von einem Objekt der

implementierenden Klasse – also der Klasse `AbhängigeKlasse` – zu einem Objekt einer konkreten zu injizierenden Klasse herzustellen. Das folgende Bild zeigt die bei Interface Injection beteiligten Klassen und Interfaces, allerdings der Einfachheit halber ohne einen Injektor und nur mit einer einzigen konkreten zu injizierenden Klasse:

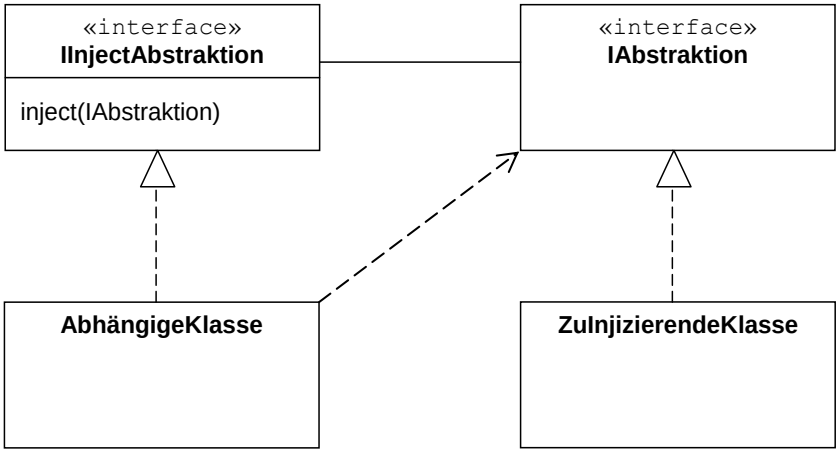

*Bild **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**-2 Klassendiagramm für Interface Injection*

Die Assoziation zwischen den beiden Interfaces im Bild **Fehler! Kein Text mit angegebener Formatvorlage im Dokument.**-2 deutet an, dass die beiden Interfaces logisch zusammengehören. M. Fowler empfiehlt in [fowioc], dass beide auch zusammen entworfen werden.

Ein Vorteil von Interface Injection ist, dass eine Klasse durch die Implementierung des entsprechenden Interface explizit kund tut, dass sie zur Injektion bereit ist. Ein weiterer Vorteil ist, dass ein solches Interface von mehreren Klassen implementiert werden kann. Damit wird der Injektor nicht von jeder einzelnen Klasse abhängig, sondern der Injektor ist nur vom Interface abhängig. Der Nachteil ist, dass **jede Klasse**, die das Interface implementiert, **von dem Interface abhängig** wird.

Aufgabe 2: Dependency Injection mit Hilfe eines einfachen IoC-Containers

In Aufgabe 2 soll ein einfacher Textprozessor betrachtet werden, dessen Aufgabe es ist eine Datei einzulesen um den Inhalt dann wieder auszugeben. Bitte ersetzen Sie die Lücken . . . durch entsprechenden Code.

Das Interface `IPrintService` deklariert eine Methode `print()`.

```
//Datei: IPrintService
public interface IPrintService {
      void print(String message);
}
```

Die Klasse `ConsolePrinter` implementiert das Interface `IPrintService`. In der Methode `print()` wird ein Text auf `stdout` ausgegeben.

```
//Datei: ConsolePrinter
public class ConsolePrinter implements IPrintService {

	@Override
	public void print(String message) {
		System.out.println(message);
	}

}
```

Das Interface `Reader` deklariert eine Methode `read()`.

```
//Datei: Reader
public interface IReader {
	String read(String path);
}
```

Die Klasse `TextReader` implementiert das Interface `IReader`. In der Methode `read()` wird der Inhalt einer txt-Datei ausgelesen und als String zurückgegeben.

```
//Datei: TextReader
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class TXTReader implements IReader{

	@Override
	public String read(String path) {
		File file = new File(path);

	  if (!file.canRead() || !file.isFile())
	      System.exit(0);

	  FileReader fr = null;
	  int c;
	  StringBuffer buff = new StringBuffer();
	  try {
	      fr = new FileReader(file);
	      while ((c = fr.read()) != -1) {
	          buff.append((char) c);
	      }
	      fr.close();

	  } catch (IOException e) {
	      e.printStackTrace();
	  }
```

```
        return buff.toString();
    }

}
```

Die Methode `process()` der Klasse `TextProcessor` erhält über den Parameter den Pfad bzw. Dateiname einer Textdatei. Diese Datei wird mit Hilfe eines konkreten Objekts vom Typ `IReader` eingelesen und durch eine konkrete Instanz des Typs `IPrintService` wieder ausgegeben.

```
//Datei: TextProcessor
public class TextProcessor {
    private Reader reader;
    private PrintService printService;

    public TextProcessor(Reader reader, PrintService printService) {
        this.reader = reader;
        this.printService = printService;
    }

    public void process(String path) {
        String text = this.reader.read(path);
        this.printService.print("Inhalt der Datei " + path + ":");
        this.printService.print(text);
    }
}
```

Eine Instanz der Klasse `IoCContainer` besitzt die beiden Methoden `register()` und `resolve()`. Die Methode `register()` ordnet einer konkreten Implementierung den Abhängigkeitstyp zu. Die Methode `resolve()` dient dazu, einen Typ unter automatischer Auflösung der Abhängigkeiten zu instanziieren. Dies geschieht hier mit Hilfe von **Reflection.** Bei diesem einfachen IoC-Container soll immer der erste Konstruktor einer Klasse ausgewählt werden. Durch die foreach-Schleife soll über die Parameter des Konstruktors iteriert werden und anschließend sollen diese erzeugt werden. Abschließend wird das zu erstellende Objekt unter Auflösung der Abhängigkeiten instanziiert und zurückgegeben.

```
//Datei: IoCContainer
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.Hashtable;

public class IoCContainer {
    // Tabelle zur Ablage der Konfiguration
    Hashtable<Class<?>, Class<?>> typeTable = new Hashtable<Class<?>,
Class<?>>();

    // Methode um Abhängigkeiten zu konfigurieren
    public void register(Class<?> dependency, Class<?> implementation) {
        // Konkrete Implementierung wird Abhänigkeitstyp zugeordnet
        typeTable.put(dependency, implementation);
    }
```

```
	// Methode um einen Typ unter automatischer Auflösung der
	Abhängigkeiten zu instanzieren
	public Object resolve(Class classWithDependencies) throws
	InstantiationException, IllegalAccessException,
	IllegalArgumentException, InvocationTargetException {

	// Ersten Konstruktor der Klasse auswählen (Achtung: Einschränkung!)
	Constructor ctr = classWithDependencies.getConstructors()[0];

	// Liste für die Übergabe der instanzierten Abängigkeiten an den
	//Konstruktor der Klasse
	ArrayList<Object> initargs = new ArrayList<Object>();

	// Über die Parameter des gewählten Konstruktors iterieren
	for(Class parameter : ctr.getParameterTypes()) {
	// Parameter instanzieren (Achtung: Keine rekursive Auflösung!
	//Einschränkung!)
		Object arg = . . .
		// Parameterobjekt in Liste aufnehmen
		. . . .
		}

		// Klasse mit automatisch aufgelösten Abhängigkeiten instanzieren
		und zurückgeben
		return . . .
	}
}
```

Die `Main()`-Methode der Klasse `Main` instanziiert einen IoC-Container. Anschließend werden die Abhängigkeiten konfiguriert und ein mit Hilfe des Containers das konkrete Objekt `Textprocessor` erstellt.

```
//Datei: Main
import java.lang.reflect.InvocationTargetException;


public class Main {

	public static void main(String[] args) throws IllegalArgumentException,
InstantiationException, IllegalAccessException, InvocationTargetException {
	// Container instanzieren
		...

	// Abhängigkeiten konfigurieren
	container.register(IPrintService.class,ConsolePrinter.class);
		. . .

     // Objekt unter automatischer Auflösung der Abhängigkeiten anlegen
     TextProcessor processor = (TextProcessor). . .

    // Objekt benutzen
   processor.process("BeispielInput.txt");
    }
```

```
}
```

Zusatzaufgabe: Dependency Injection – Bessere Version –

Der IoC-Container enthält einige Einschränken:

- Der Container ist nicht als Singleton implementiert, was eine verteilte Anwendbarkeit einschränkt.
- Der Container versucht nicht einen „auflösbaren“ Konstruktor zu finden, sondern nimmt einfach den ersten.
- Der Container arbeitet nicht rekursiv, sondern geht davon aus, dass die Parameter direkt instanzierbar sind.
- Der Container wird im Quellcode konfiguriert. Besser wäre eine Auslagerung der Konfiguration in eine Property-Datei.

Verbessern Sie den IoC-Container durch Beseitigung der Einschränkungen.